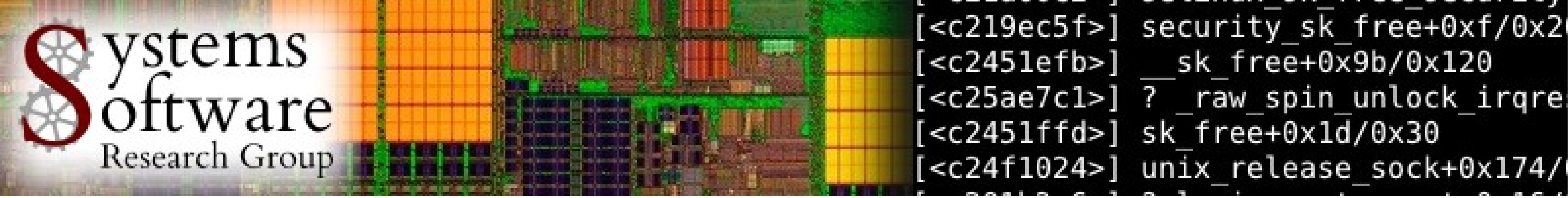


```
[<c219ec5f>] security_sk_free+0xf/0x2
[<c2451efb>] __sk_free+0x9b/0x120
[<c25ae7c1>] ? _raw_spin_unlock_irqre
[<c2451ffd>] sk_free+0x1d/0x30
[<c24f1024>] unix_release_sock+0x174/
```

# Designing, Modeling, and Optimizing Transactional Data Structures

PhD Dissertation Defense

**Ahmed Hassan**

Electrical and Computer Engineering Department
Virginia Tech

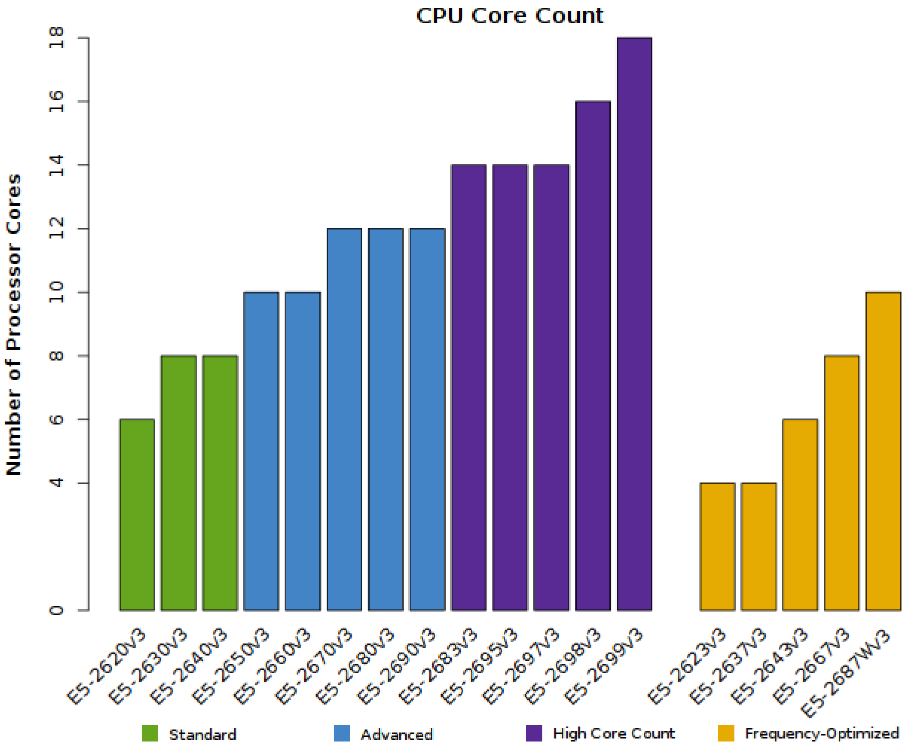
September 1, 2015

# State of the Art

- Multi-core architectures.

- Synchronization
  - Critical sections.
  - Using locks.

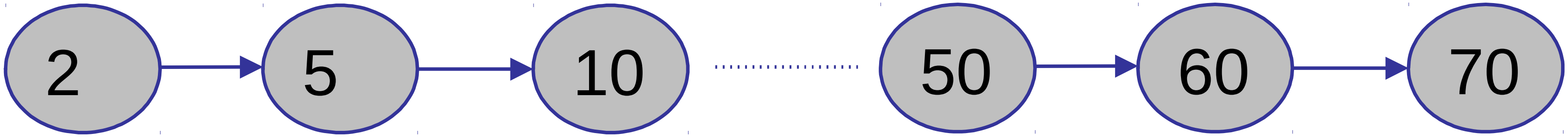
- Efficient synchronization:
  - Cache coherence protocols.
  - Atomic instructions
    (e.g. CAS operations).



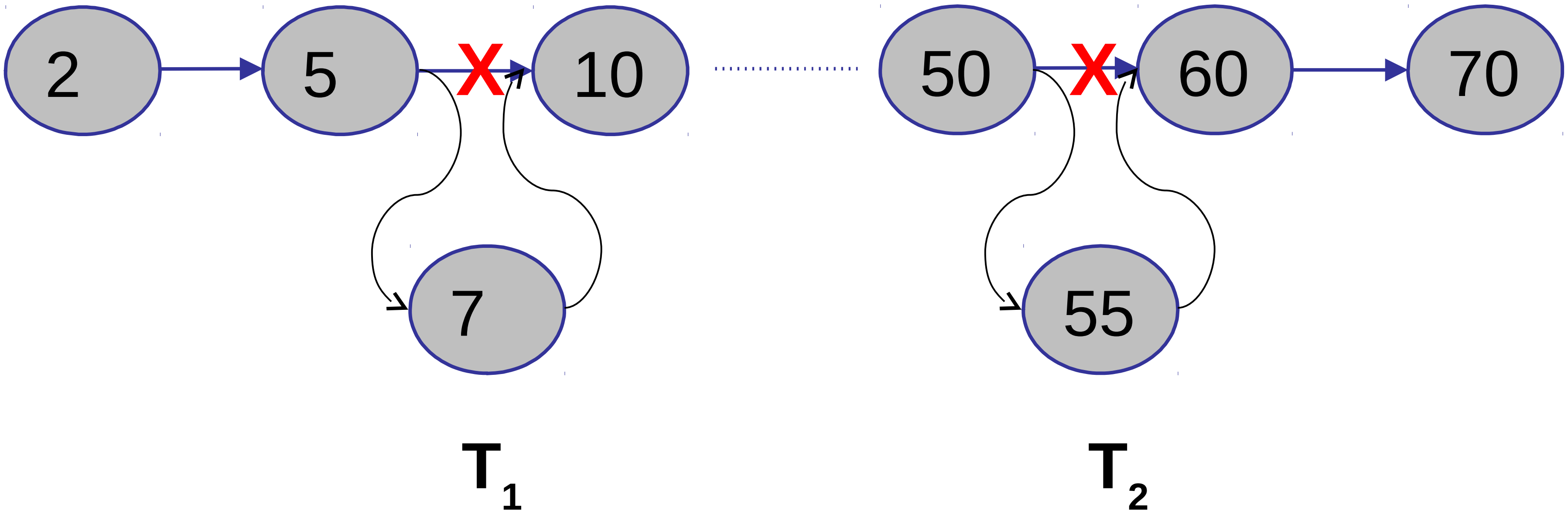
From *www.microway.com*

# Concurrent Data Structures

# Example: Linked List



# Example: Linked List



# Synchronization in Concurrent Data Structures

- Coarse-grained locking
  - Easy to implement, good for low number of small threads .
  - **But** minimizes concurrency.

- Fine-grained locking
  - Allows more concurrency.
  - **But** error prone.

- Non-Blocking Designs
  - Lock-free, obstruction-free, wait-free, …
  - Progress guarantees **But** more complex designs.

# Transactional Memory

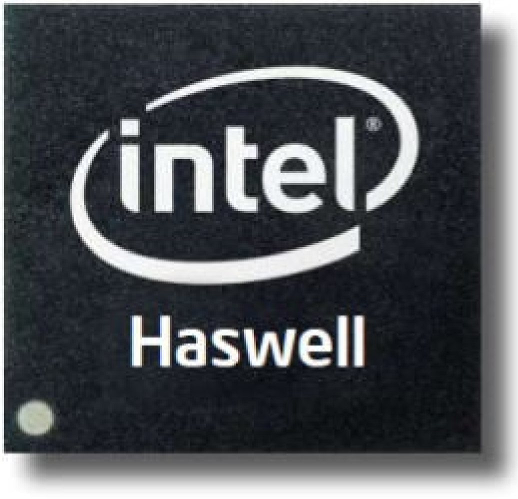
- Use an underlying TM framework to guarantee consistency, atomicity, and isolation.

```
Thread 1

@Atomic
foo1()
{
   seqList.add(5)
}
```

- Programmable (like coarse-grained locking).
- Allows concurrency (like fine-grained locking).

# Transactional Memory Gains Traction!!

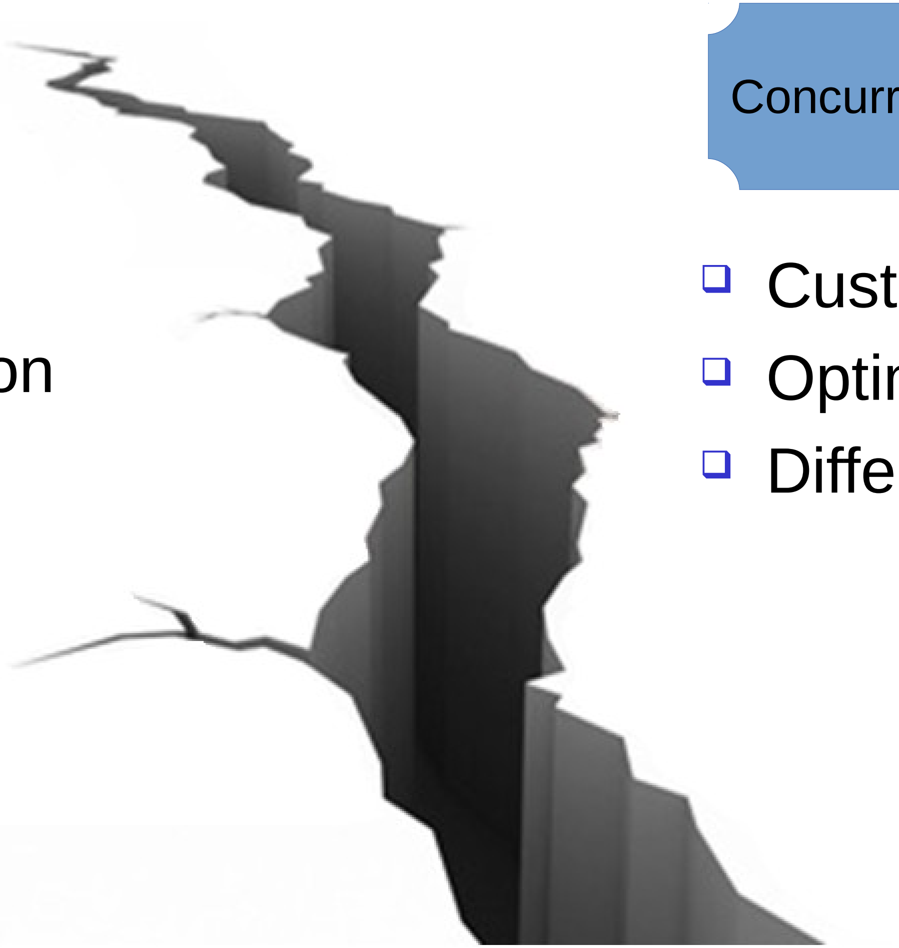
- Intel Haswell's TSX Extensions.
- IBM Power8.
- STM support in C++ and GCC.



# What about data structures?

Transactional Programming Model

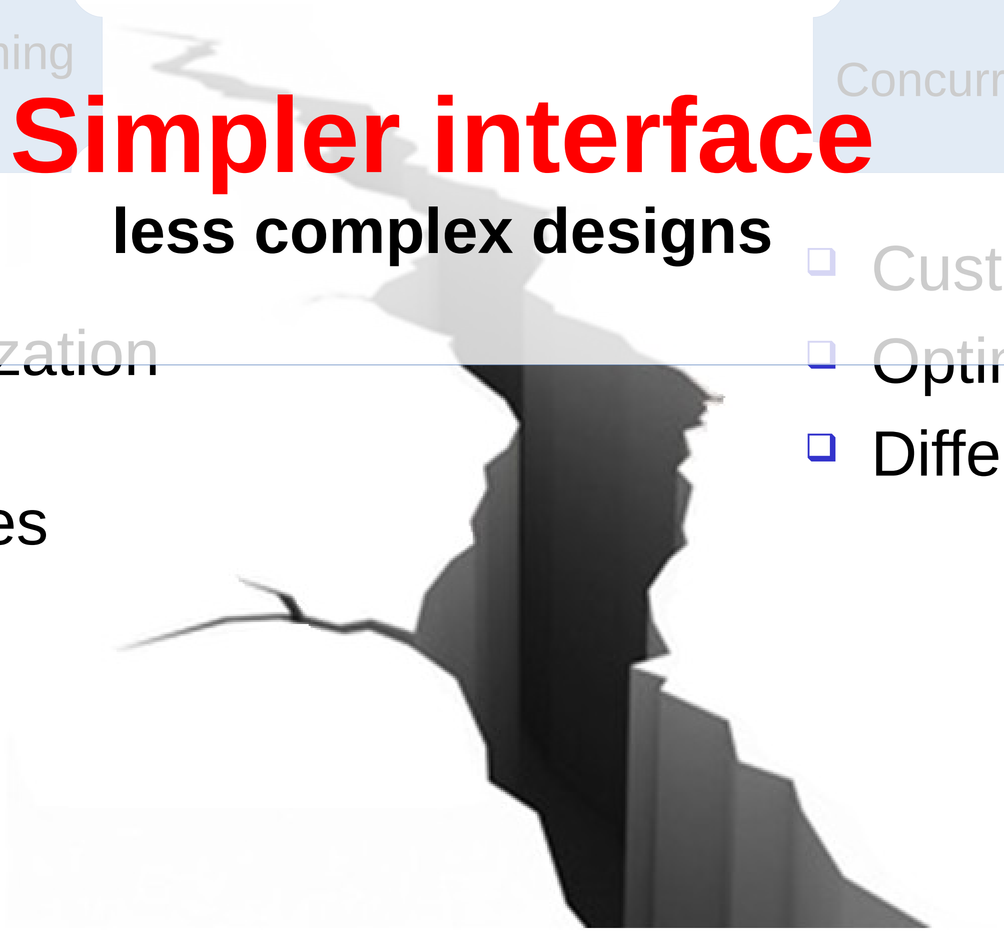
- **New** APIs
- **New** synchronization primitives.
- **New** TM libraries
- **New** compiler support.
- **New** hardware components.



Concurrent Data Structures

- Customized Designs
- Optimizations
- Different Primitives

# What about data structures?



**Simpler interface**

**less complex designs**

Transactional Programming Model

- **New** APIs
- **New** synchronization primitives.
- **New** TM libraries
- **New** compiler support.
- **New** hardware components.

Concurrent Data Structures

- Customized Designs
- Optimizations
- Different Primitives

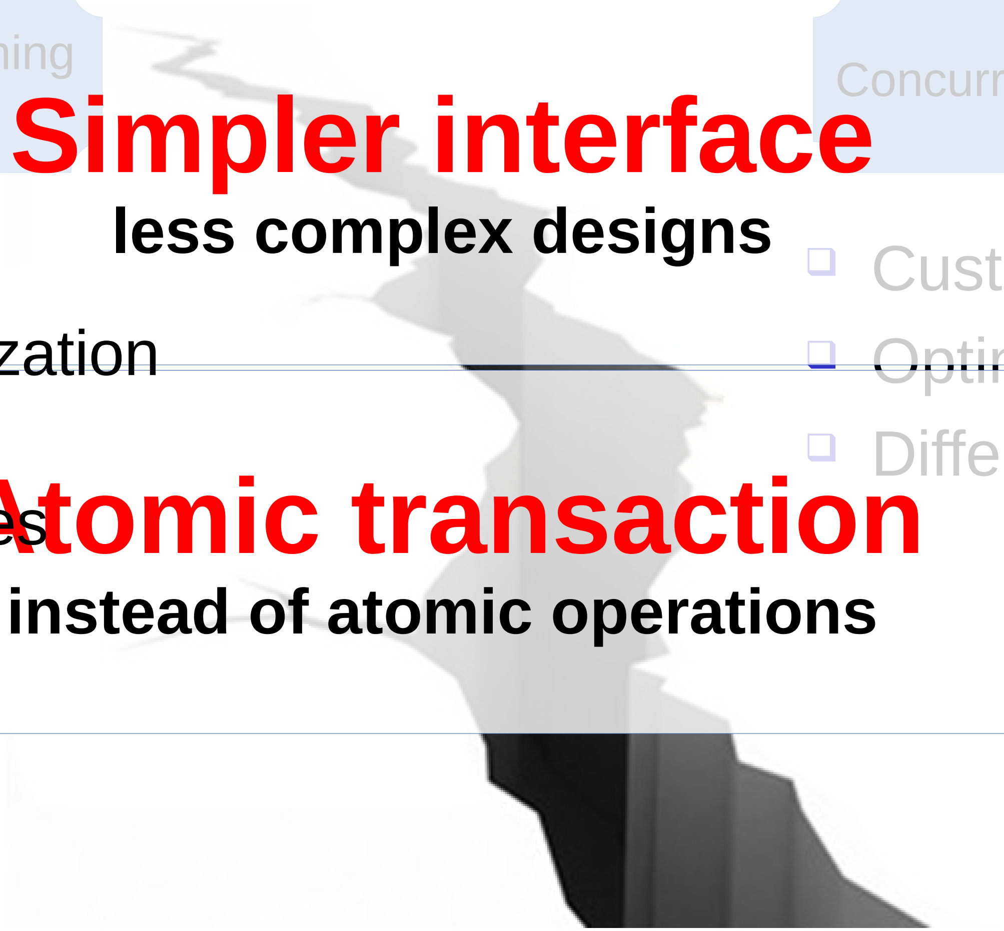
# What about data structures?

Transactional Programming Model

- **New** APIs
- **New** synchronization primitives.
- **New** TM libraries
- **New** compiler support.
- **New** hardware components.

Concurrent Data Structures

- Customized Designs
- Optimizations
- Different Primitives



**Simpler interface**

**less complex designs**

**Atomic transaction**

**instead of atomic operations**

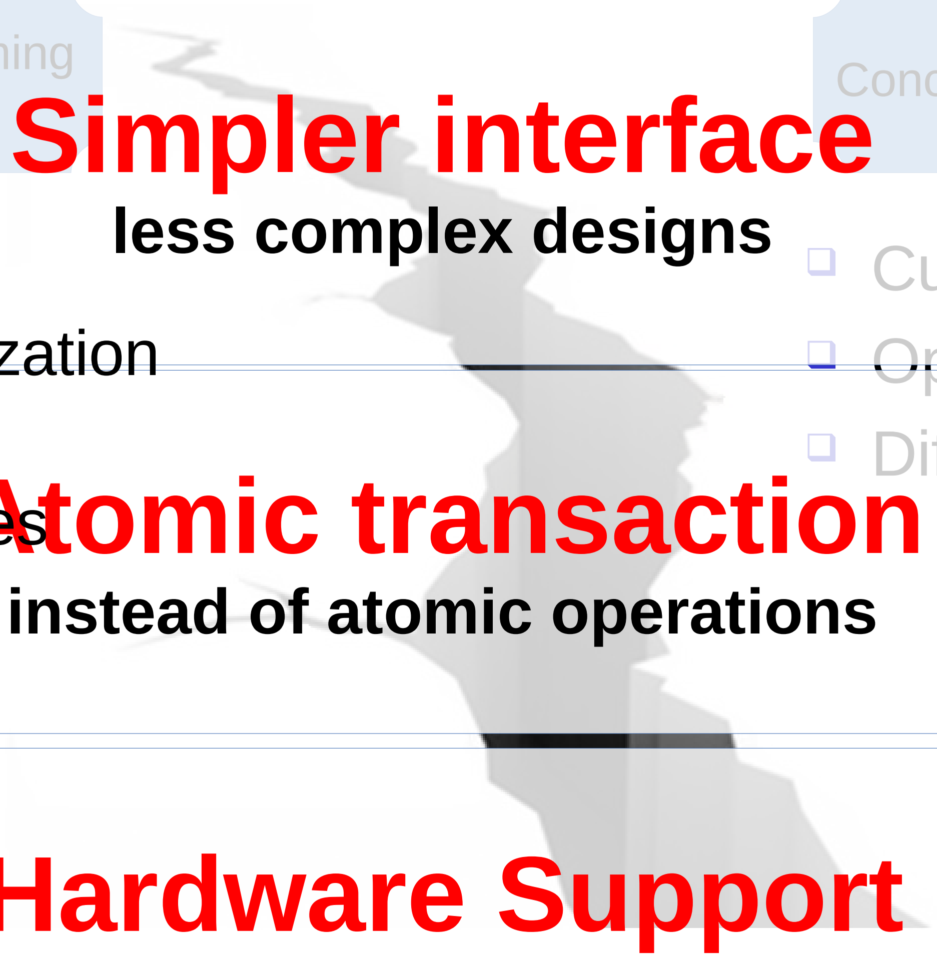
# What about data structures?

Transactional Programming Model

- **New** APIs
- **New** synchronization primitives.
- **New** TM libraries
- **New** compiler support.
- **New** hardware components.

Concurrent Data Structures

- Customized Designs
- Optimizations
- Different Primitives



**Simpler interface**

**less complex designs**

**Atomic transaction**

**instead of atomic operations**

**Hardware Support**

**possible performance improvement**

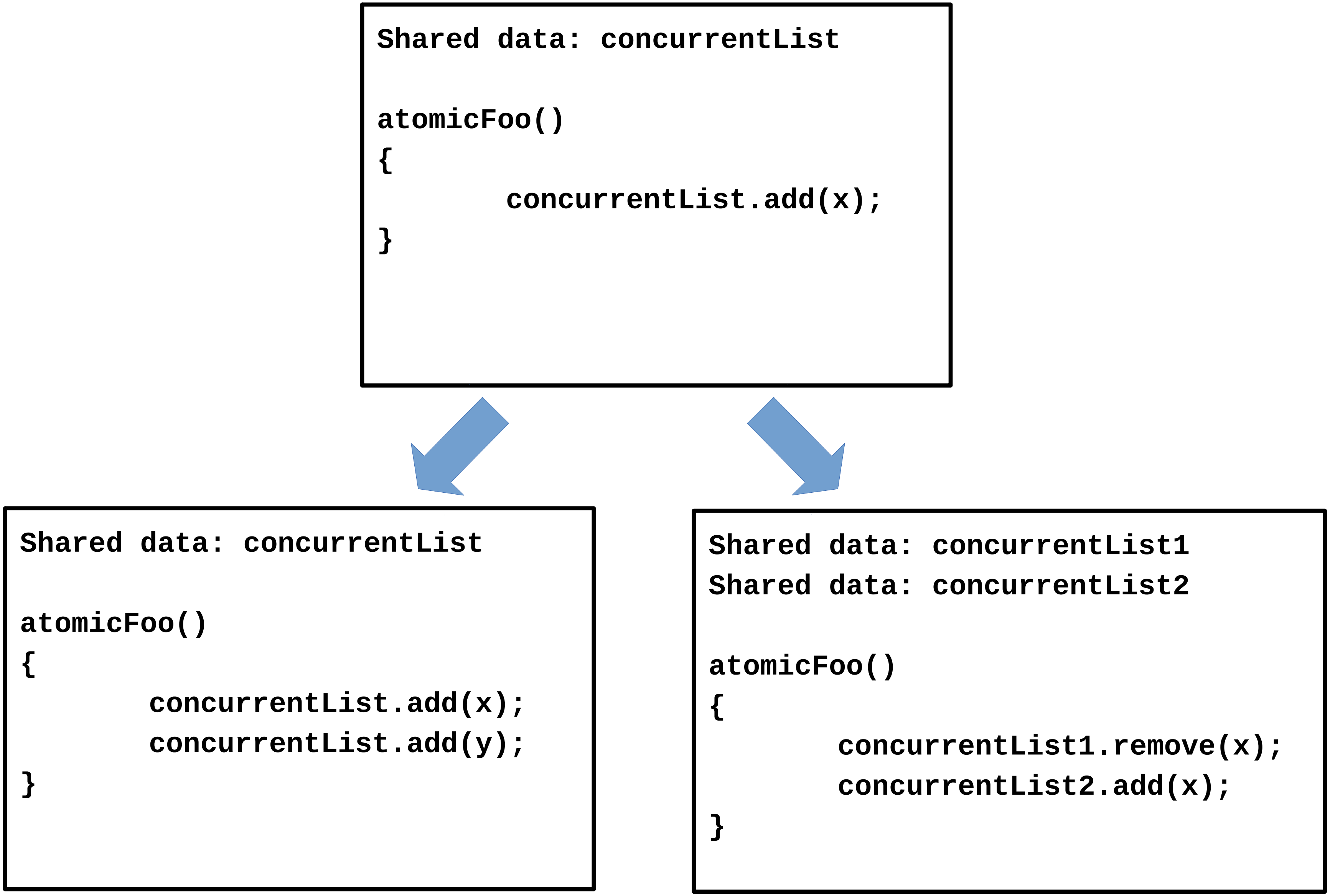
# Our Goal

## From Concurrent to Transactional Data Structures

# Challenges

- Composability.
- Integration with generic transactions.
- Modeling.

# Composability



```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
}
```

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
        concurrentList.add(y);
}
```

```
Shared data: concurrentList1
Shared data: concurrentList2

atomicFoo()
{
        concurrentList1.remove(x);
        concurrentList2.add(x);
}
```

# Composability

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
}
```

**Modify the design of concurrentList?**

**More complex designs**

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
        concurrentList.add(y);
}
```

```
Shared data: concurrentList1
Shared data: concurrentList2

atomicFoo()
{
        concurrentList1.remove(x);
        concurrentList2.add(x);
}
```

# Composability

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
}
```

**Modify the design of concurrentList?**

**More complex designs**

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
        concurrentList.add(y);
}
```

```
Shared data: concurrentList1
Shared data: concurrentList2

atomicFoo()
{
        concurrentList1.remove(x);
        concurrentList2.add(x);
}
```

**Transactional Memory?**

**Lose optimizations of concurrentList**

# Integration

```
Shared data: concurrentList

atomicFoo()
{
        If(concurrentList.add(x))
                n1++;
        Else
                n2++;
}
```

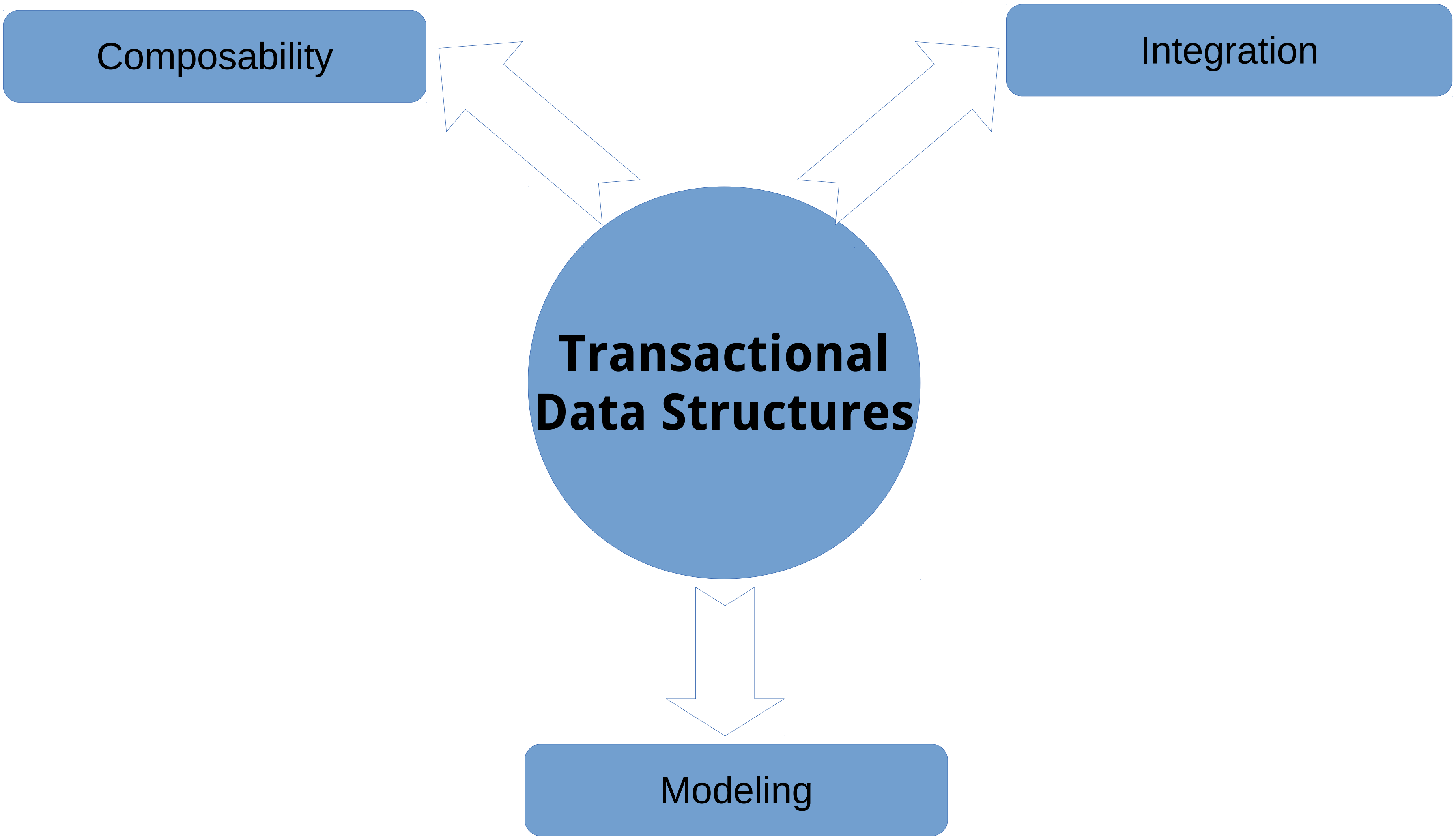
# Modeling

- Different Designs and Implementations
  - Different ad-hoc approaches for proving correctness.

- Is there a unified model for concurrent data structures?
  - General enough
  - Easy to use
  - Includes composability and integration

# Our Contributions

# Our Contributions



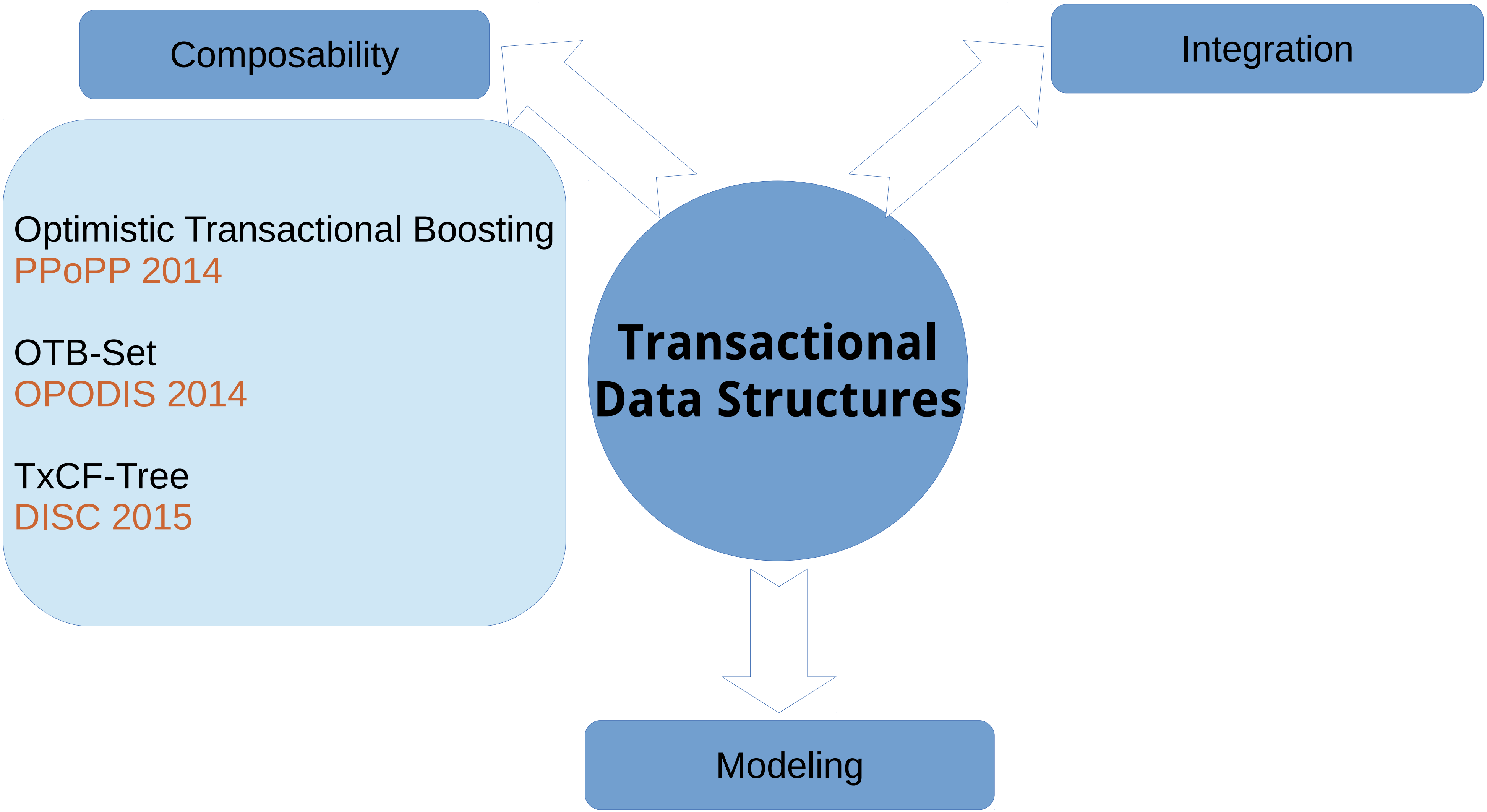
Composability

Integration

Transactional Data Structures

Modeling

# Our Contributions



Transactional Data Structures

Composability

Optimistic Transactional Boosting
PPoPP 2014

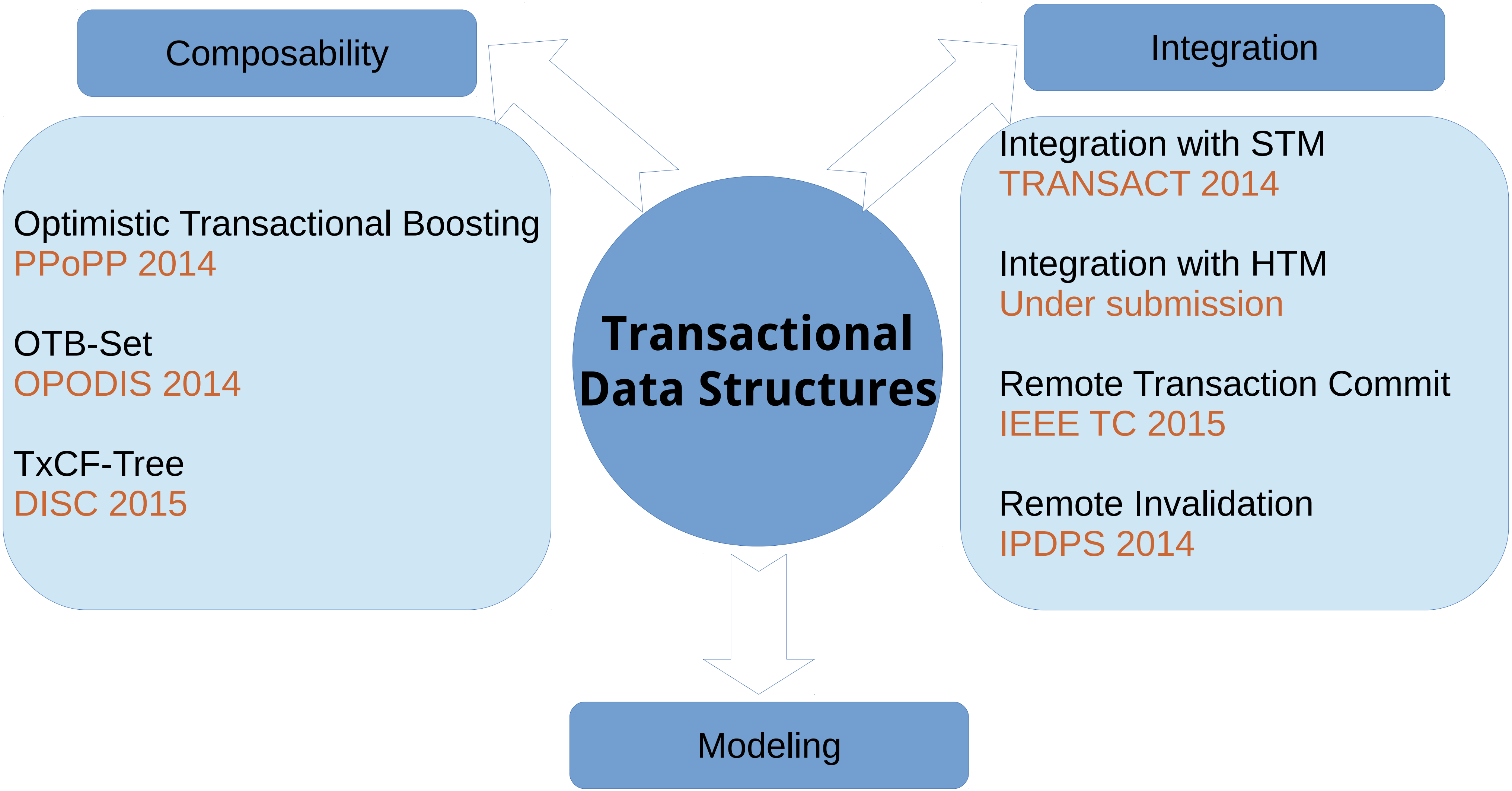
OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Integration

Modeling

# Our Contributions



Composability

Optimistic Transactional Boosting
PPoPP 2014

OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Transactional Data Structures

Integration

Integration with STM
TRANSACT 2014

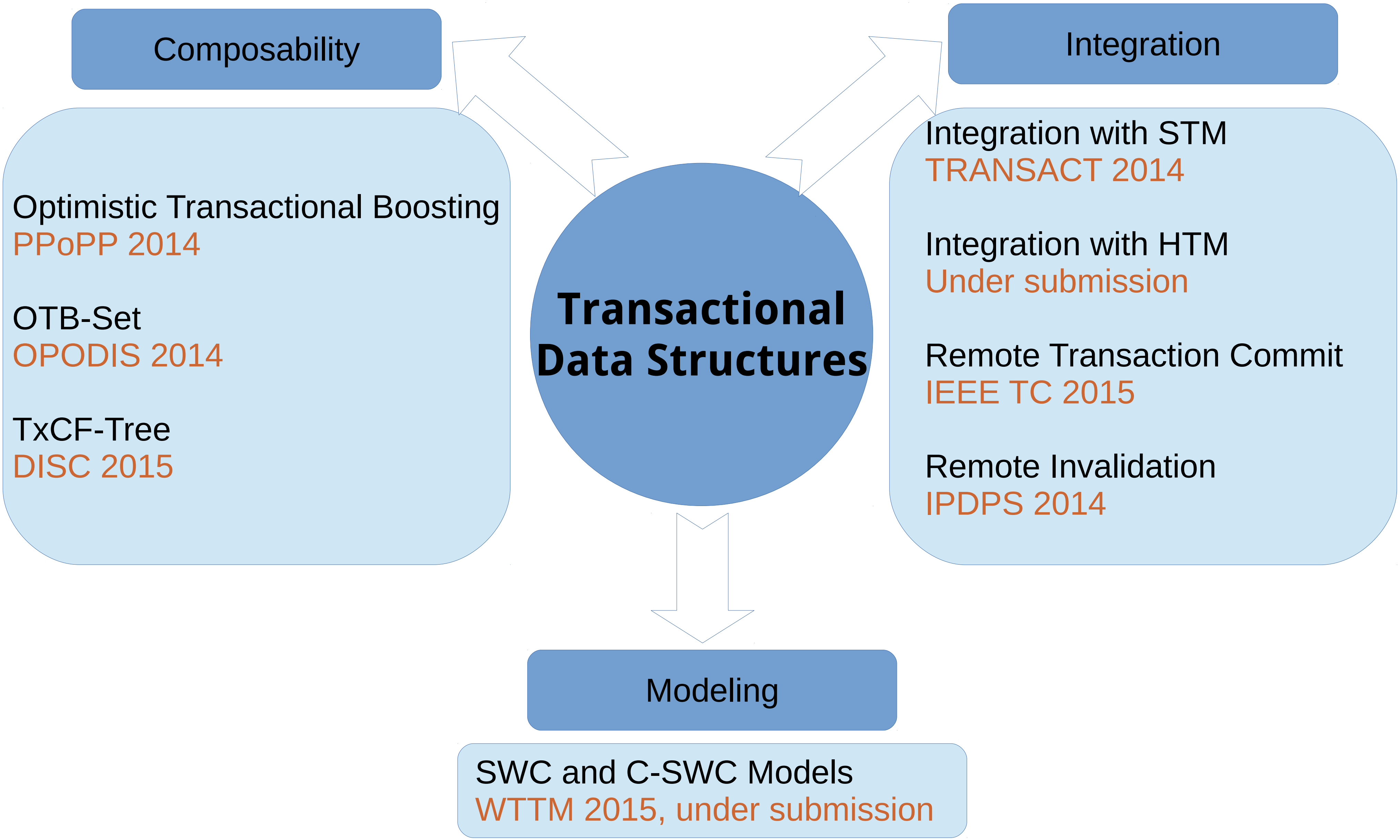
Integration with HTM
Under submission

Remote Transaction Commit
IEEE TC 2015

Remote Invalidation
IPDPS 2014

Modeling

# Our Contributions



Transactional Data Structures

Composability

Optimistic Transactional Boosting
PPoPP 2014

OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Integration

Integration with STM
TRANSACT 2014

Integration with HTM
Under submission

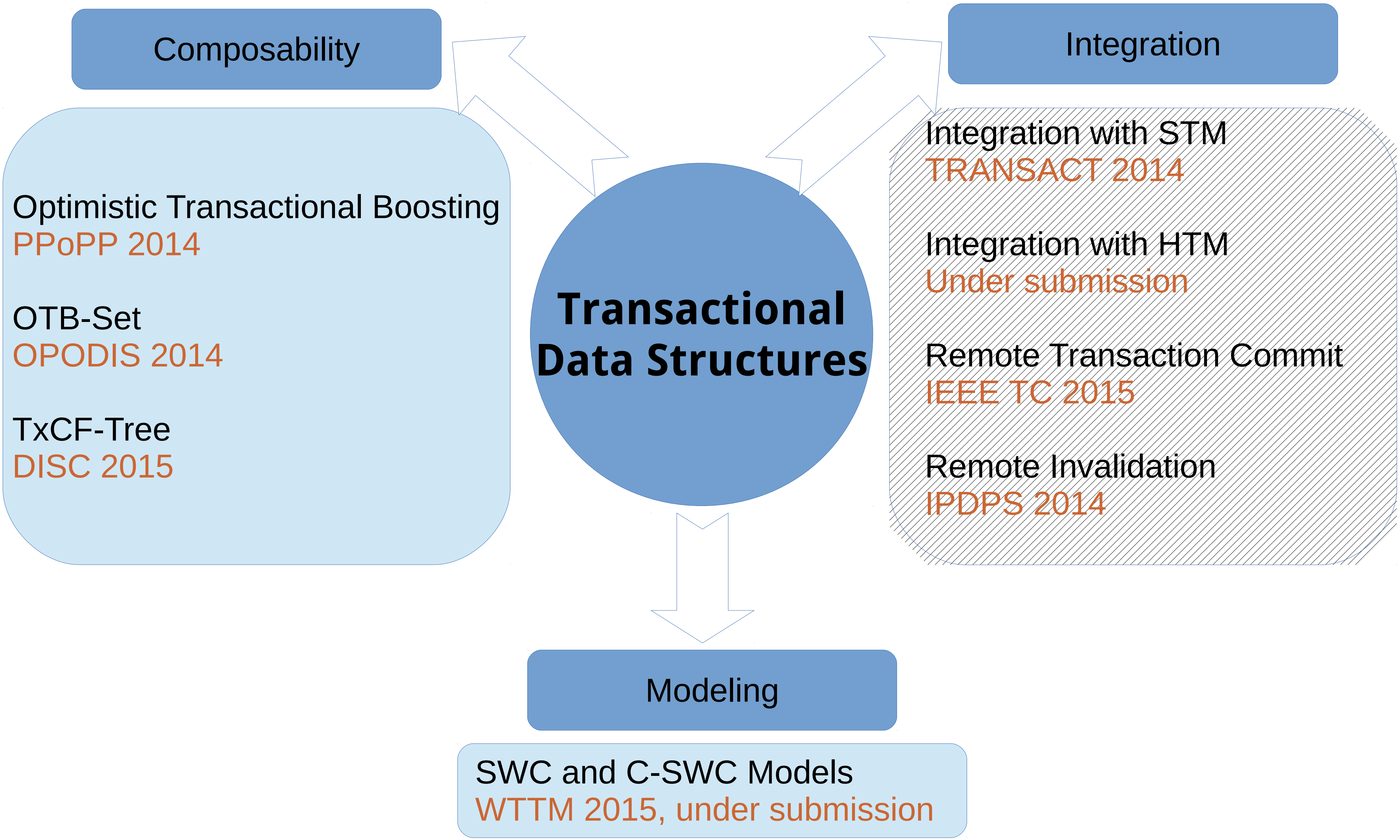
Remote Transaction Commit
IEEE TC 2015

Remote Invalidation
IPDPS 2014

Modeling

SWC and C-SWC Models
WTTM 2015, under submission

# Our Contributions



Transactional Data Structures

Composability

Optimistic Transactional Boosting
PPoPP 2014

OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Integration

Integration with STM
TRANSACT 2014

Integration with HTM
Under submission

Remote Transaction Commit
IEEE TC 2015

Remote Invalidation
IPDPS 2014

Modeling

SWC and C-SWC Models
WTTM 2015, under submission

# Past and Related Work

# Past and Related Work

- Composability and Integration
  - Transactional Memory.
  - Transactional Boosting.
- Modeling
  - SWMR Model

# Past and Related Work

- Composability and Integration
  - Transactional Memory.
  - Transactional Boosting.
- Modeling
  - SWMR Model

# Transactional Memory

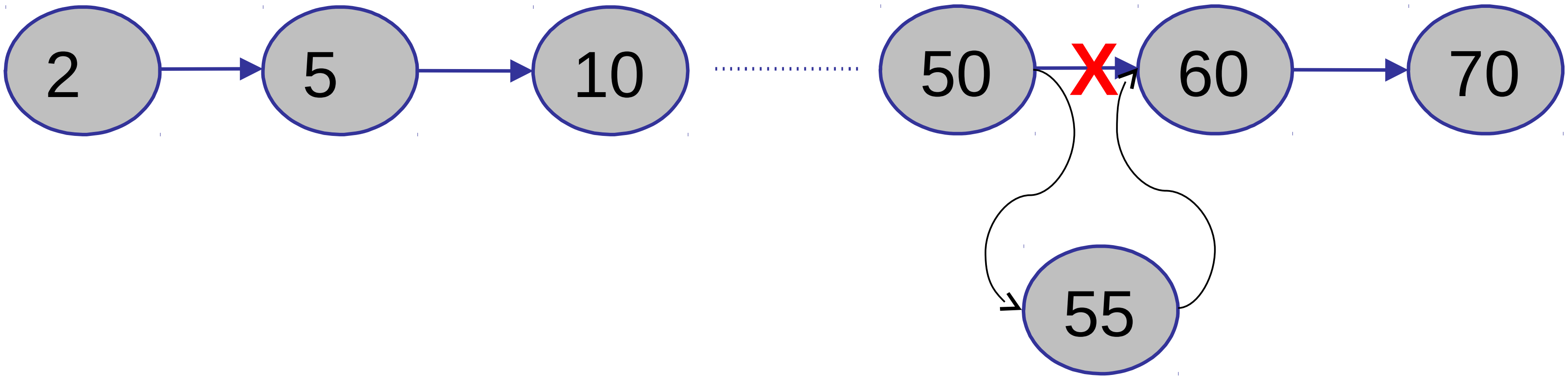
- Software Transactional Memory (STM)
  - SW meta-data (e.g. read-sets and write-sets) on the current HW.

- Hardware Transactional Memory (HTM)
  - New HW (modify cache coherency protocols).

- Hybrid Transactional Memory (Hybrid TM)
  - HTM transactions fall-back to STM

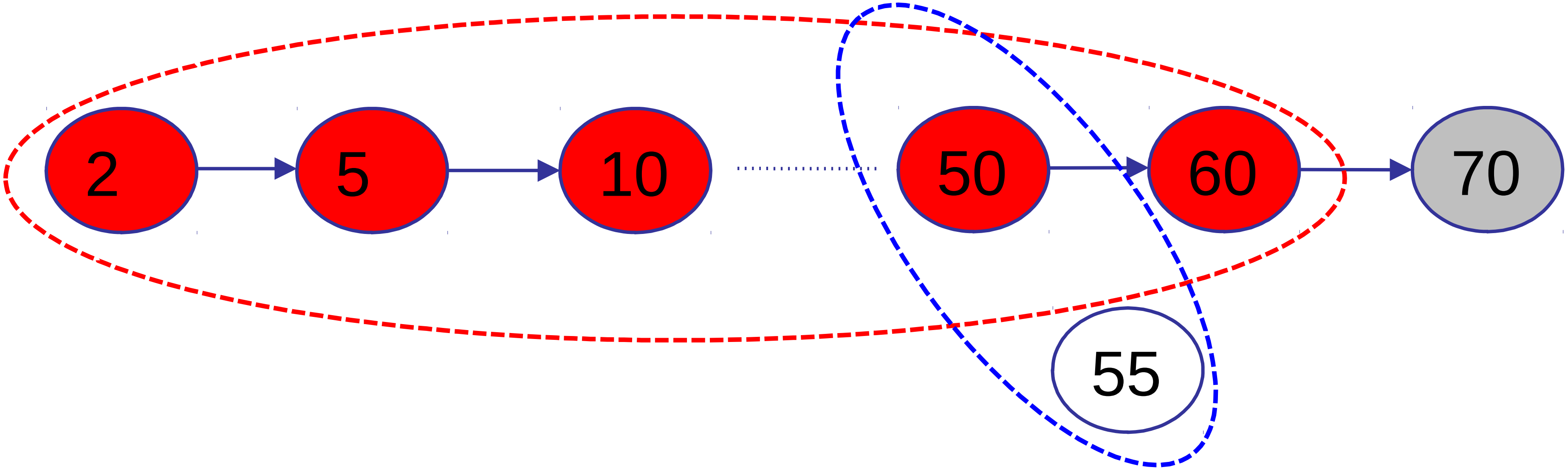
# TM Limitation: False Conflict

Example: Linked list (Insert “55”)



# TM Limitation: False Conflict

Example: Linked list (Insert “55”)

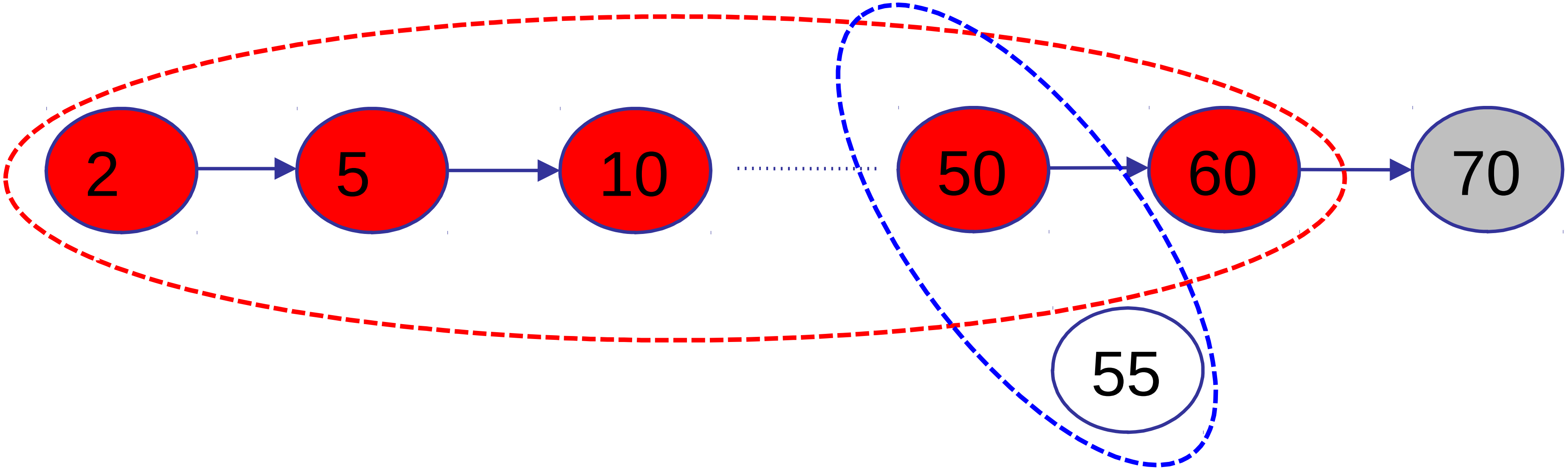


All “red” nodes are in the read-set

“50” and “55” are in the write-set

# TM Limitation: False Conflict

Example: Linked list (Insert “55”)

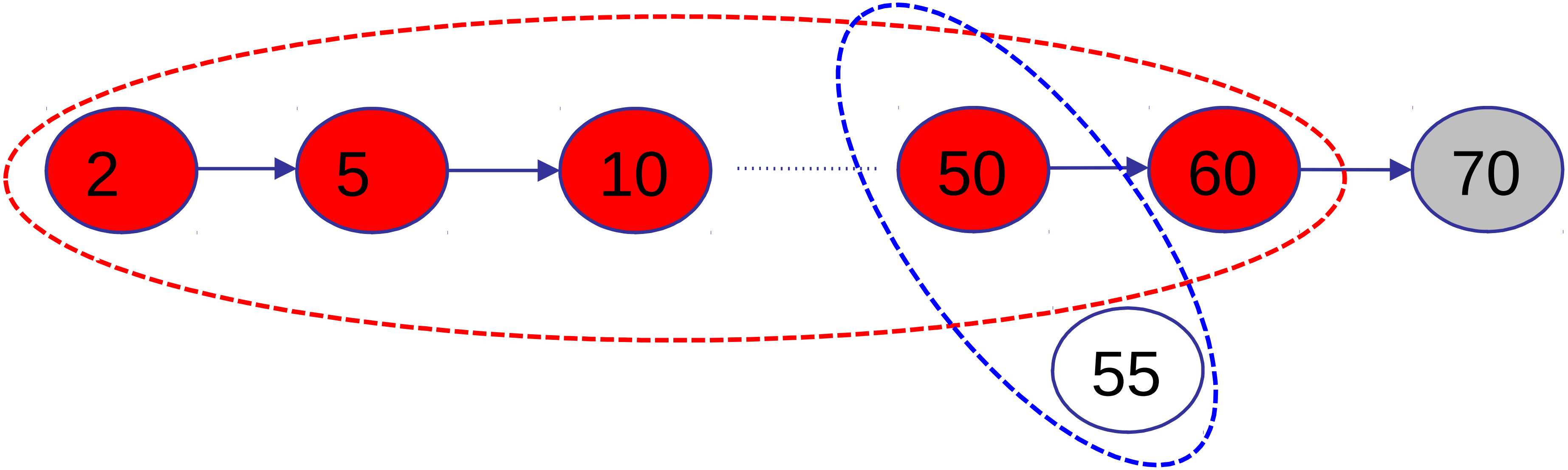


All “red” nodes are in the read-set
“50” and “55” are in the write-set

What if a concurrent transaction deletes “5”??

# TM Limitation: False Conflict

Example: Linked list (Insert “55”)



All “red” nodes are in the read-set

“50” and “55” are in the write-set

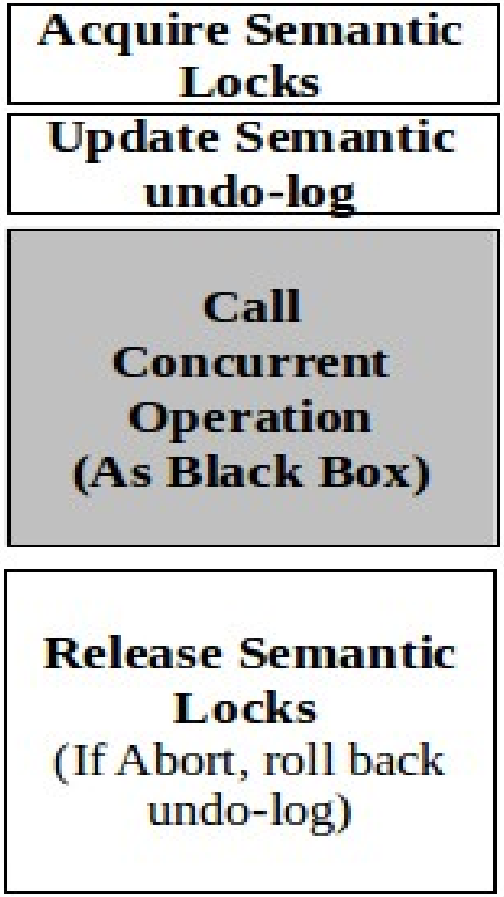
What if a concurrent transaction deletes “5”??

False Conflict

# Transactional Boosting

- Convert highly concurrent data structures to be transactional.

- Composable (like STM)
- And efficient (like lazy/lock-free linked-list)



Acquire Semantic Locks

Update Semantic undo-log

Call Concurrent Operation (As Black Box)

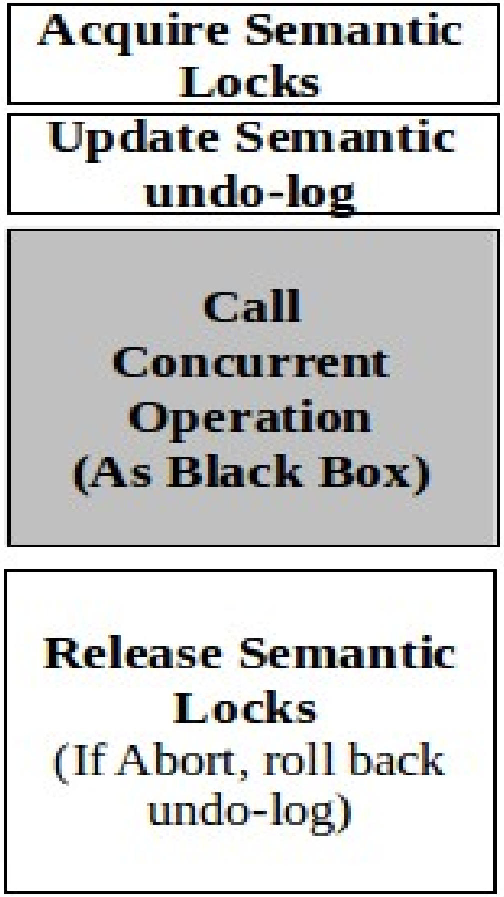
Release Semantic Locks (If Abort, roll back undo-log)

# Transactional Boosting

- Convert highly concurrent data structures to be transactional.

- Composable (like STM)
- And efficient (like lazy/lock-free linked-list)

- Issues:
  - Eager locking.
  - Inverse operations.
  - Black-box concurrent data structure.
  - No Straightforward Integration



Acquire Semantic Locks

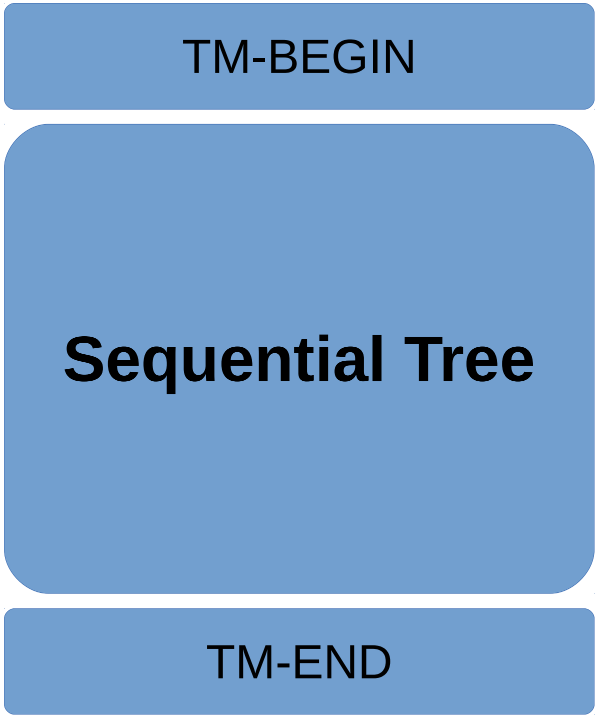
Update Semantic undo-log

Call Concurrent Operation (As Black Box)

Release Semantic Locks (If Abort, roll back undo-log)

# Optimistic Transactional Boosting

# Past Solutions

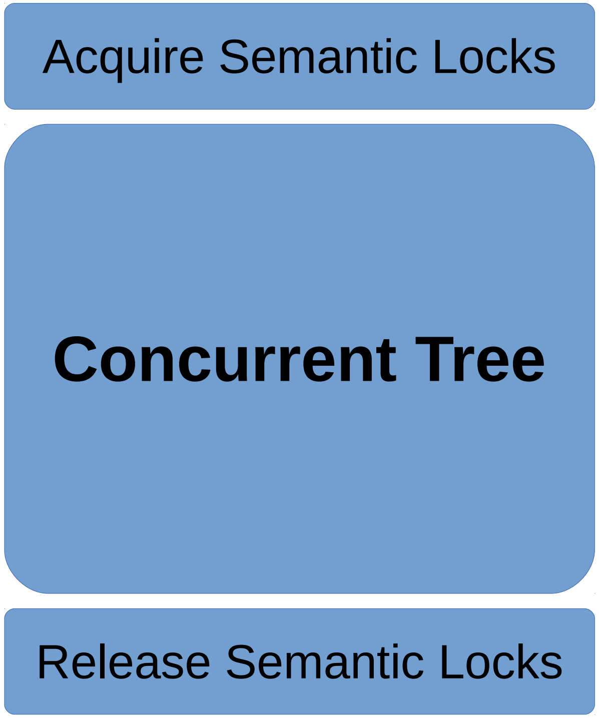


TM-BEGIN

**Sequential Tree**

TM-END

Transactional Memory



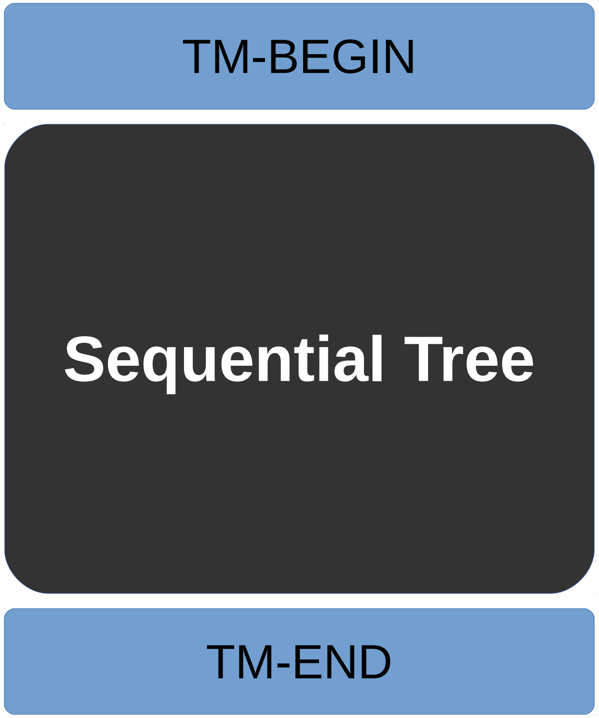
Acquire Semantic Locks

**Concurrent Tree**

Release Semantic Locks

Transactional Boosting

# Past Solutions

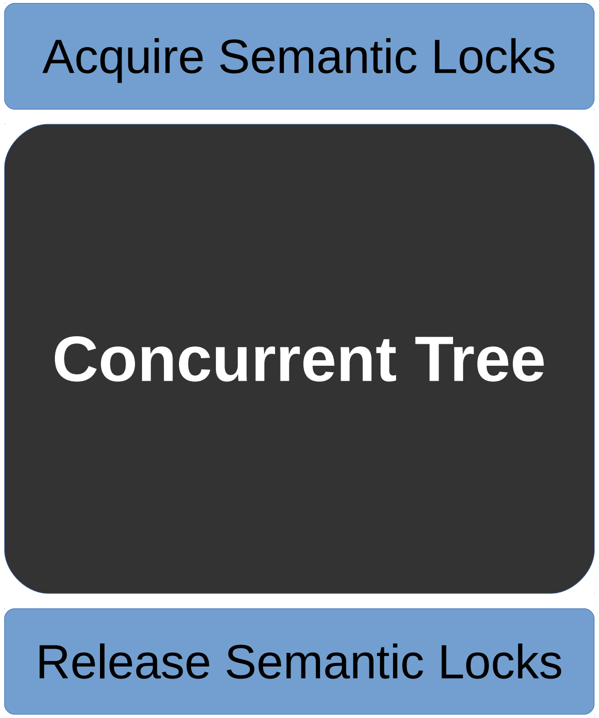


TM-BEGIN

Sequential Tree

TM-END

Transactional Memory



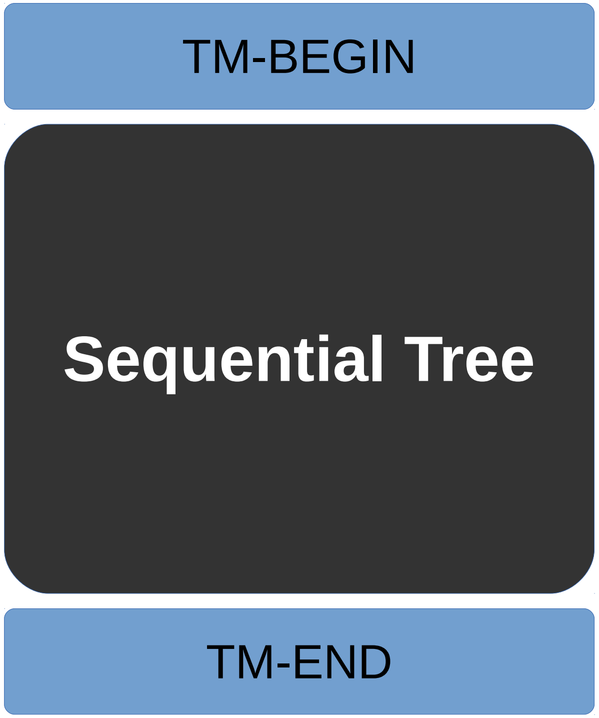
Acquire Semantic Locks

Concurrent Tree

Release Semantic Locks

Transactional Boosting

# Past Solutions

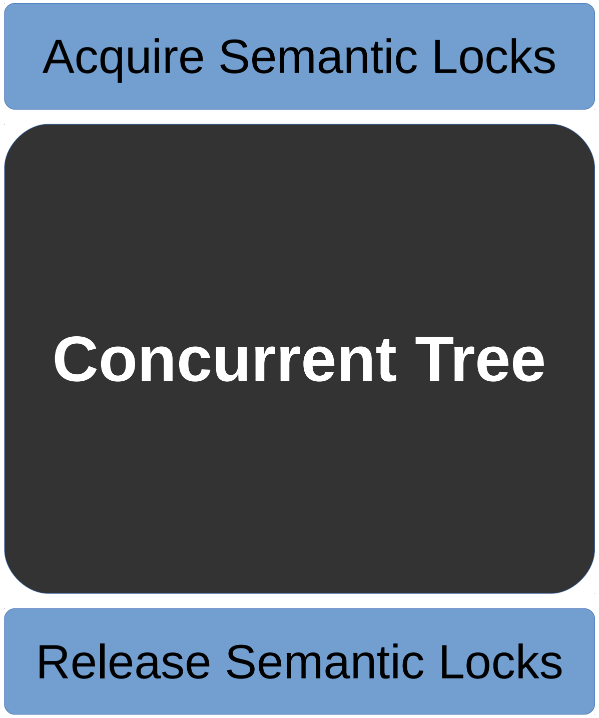


TM-BEGIN

Sequential Tree

TM-END

Transactional Memory



Acquire Semantic Locks

Concurrent Tree

Release Semantic Locks

Transactional Boosting

General, BUT not optimized.

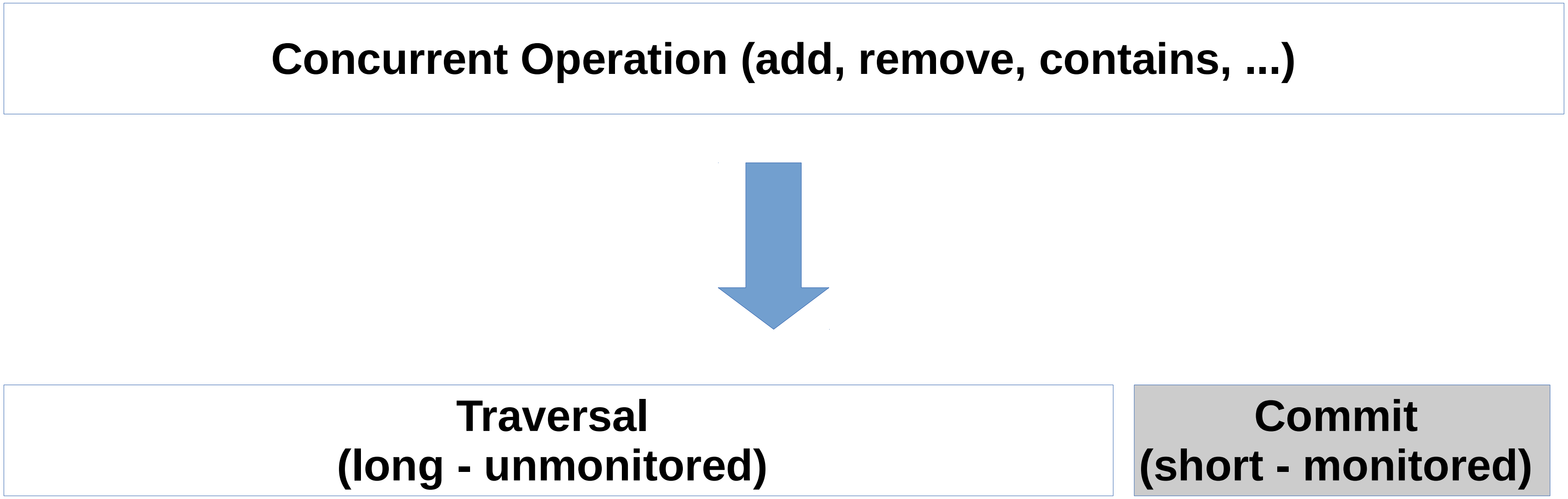
# OTB's Three Guidelines

- G1: Split operation

**Concurrent Operation (add, remove, contains, ...)**

# OTB's Three Guidelines

- G1: Split operation



Concurrent Operation (add, remove, contains, ...)

Traversal
(long - unmonitored)

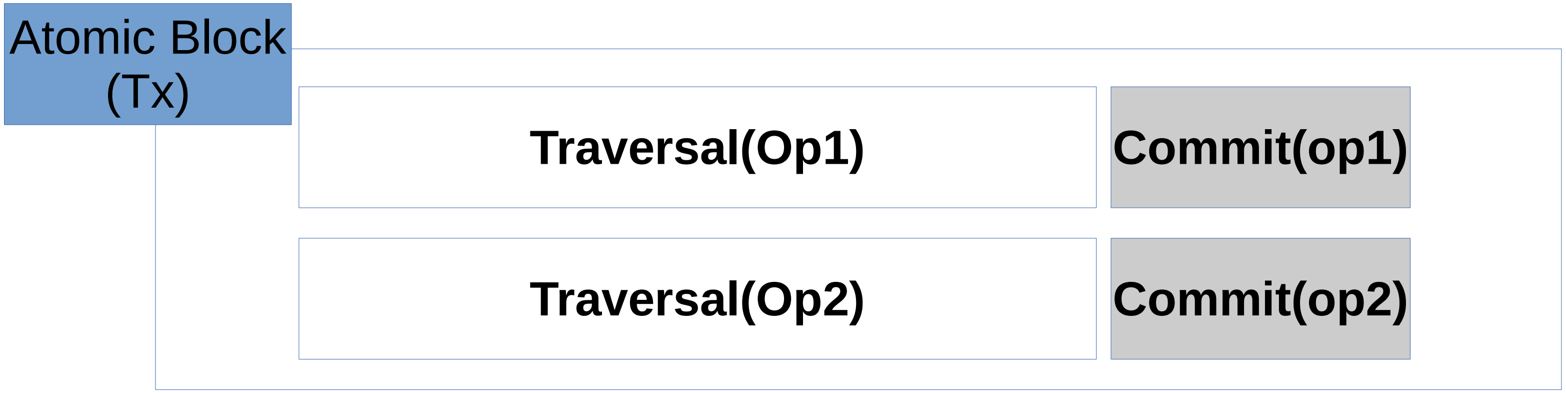
Commit
(short - monitored)

# OTB's Three Guidelines

- G2: Compose phases.

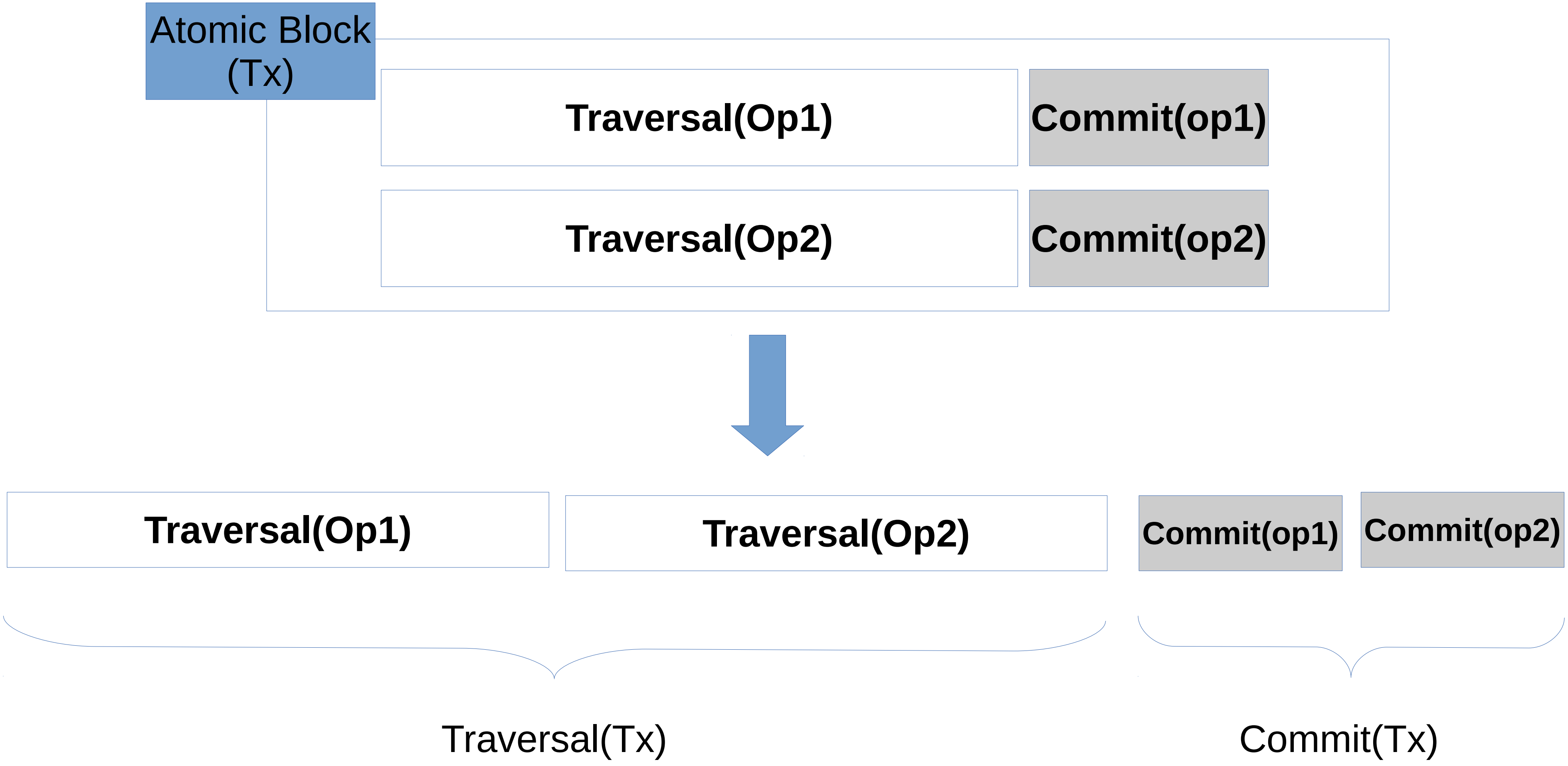
# OTB's Three Guidelines

- G2: Compose phases



# OTB's Three Guidelines

- G2: Compose phases



# OTB's Three Guidelines

- G3: Optimize

# OTB's Three Guidelines

- G3: Optimize
  - Specific to each data structure.

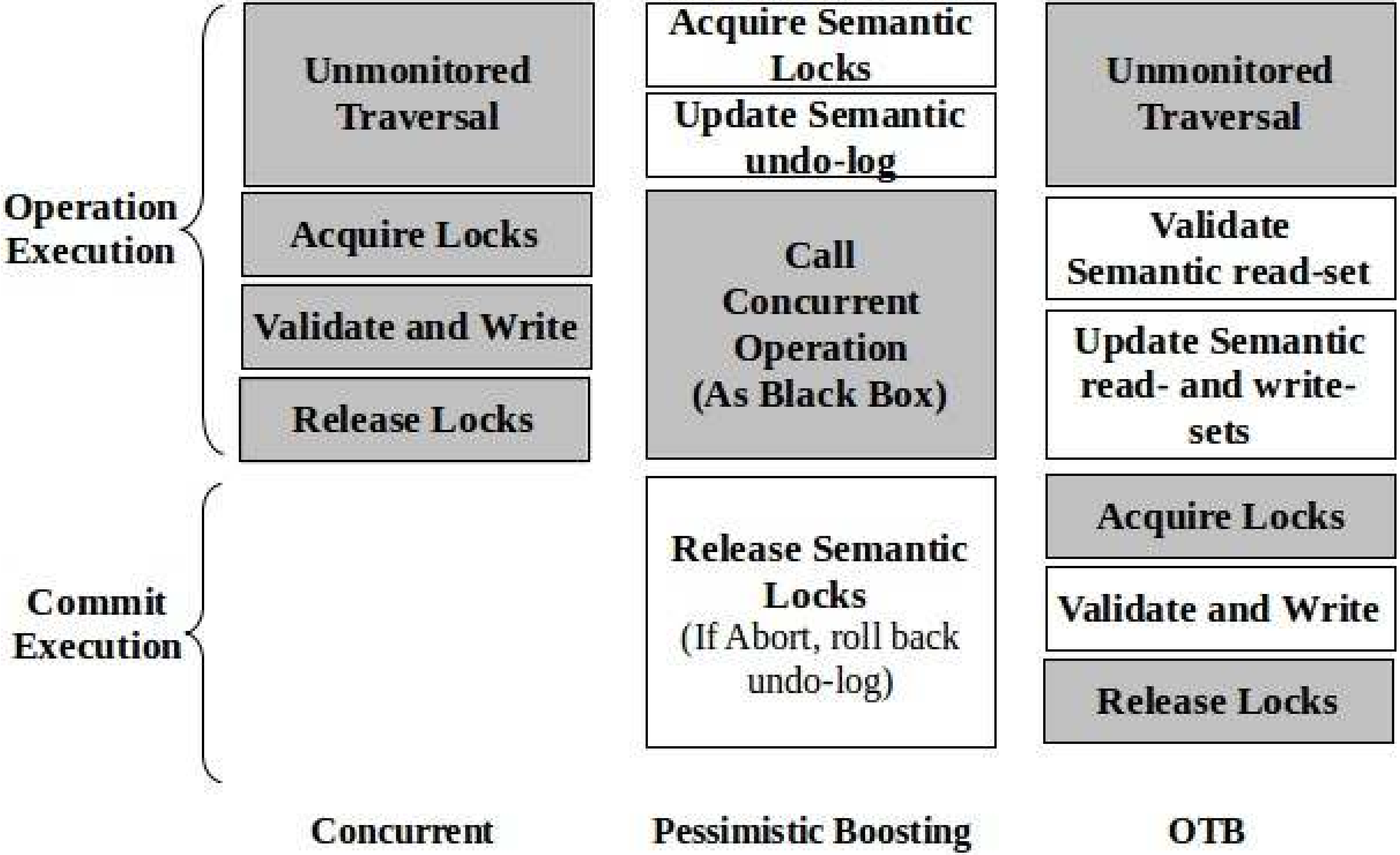
- Our Contribution:
  - Linked-list-based Set.
  - Skip-list-based Set.
  - Skip-list-based Priority Queue.
  - Balanced Tree

# OTB's Three Guidelines

- G3: Optimize
  - Specific to each data structure.

- Our Contribution:
  - Linked-list-based Set.
  - Skip-list-based Set.
  - Skip-list-based Priority Queue.
  - Balanced Tree

# Lazy Vs Boosting Vs Optimistic Boosting



Operation Execution

Commit Execution

Concurrent

Unmonitored Traversal

Acquire Locks

Validate and Write

Release Locks

Pessimistic Boosting

Acquire Semantic Locks

Update Semantic undo-log

Call Concurrent Operation (As Black Box)

Release Semantic Locks (If Abort, roll back undo-log)

OTB

Unmonitored Traversal

Validate Semantic read-set

Update Semantic read- and write-sets

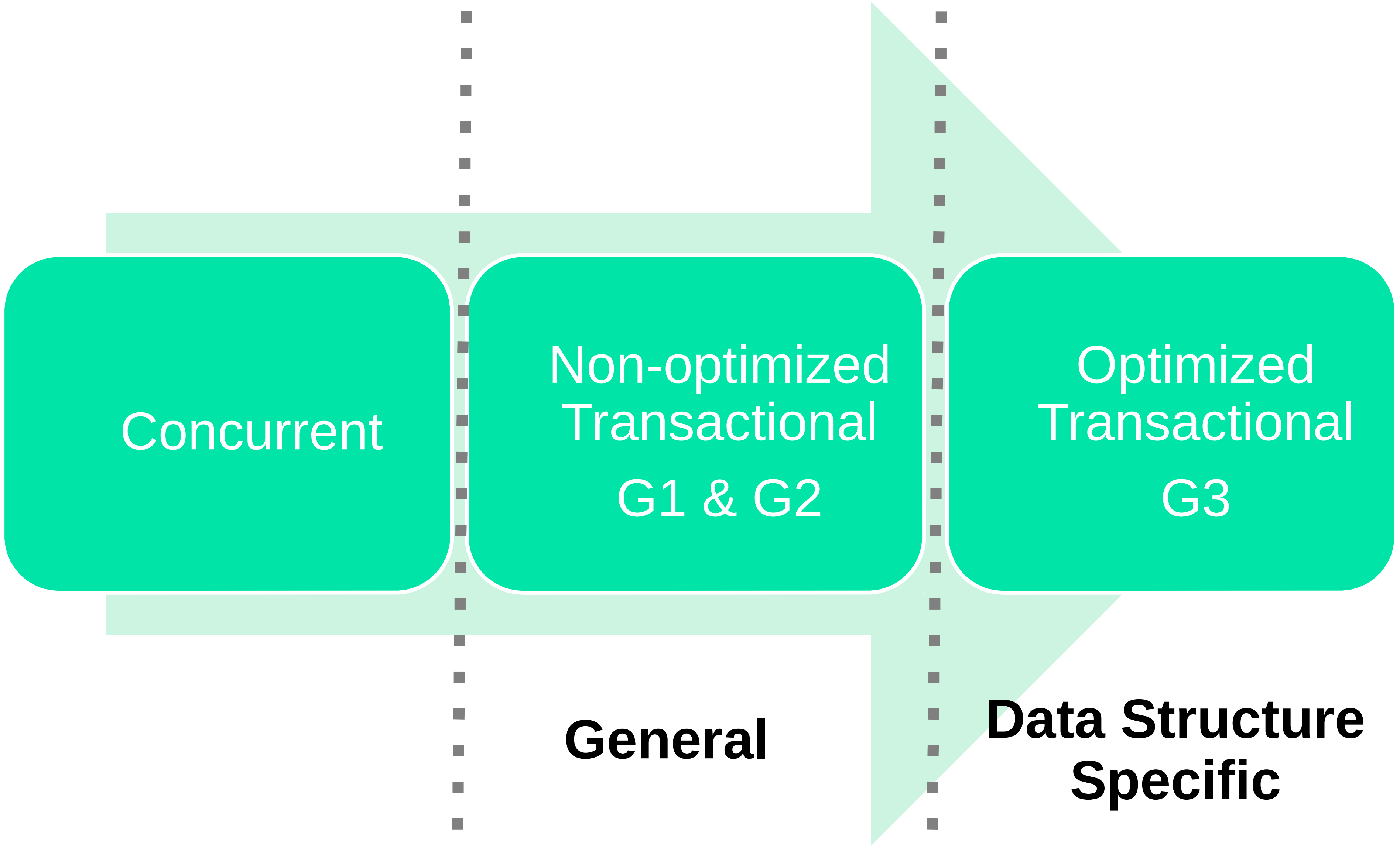
Acquire Locks

Validate and Write

Release Locks

# Example
# Bootsing “lazy” concurrent linked list

# OTB Methodology



Concurrent

Non-optimized Transactional

G1 & G2

Optimized Transactional

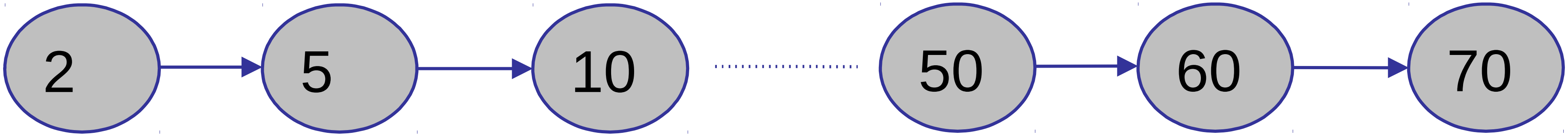
G3

**General**

**Data Structure Specific**

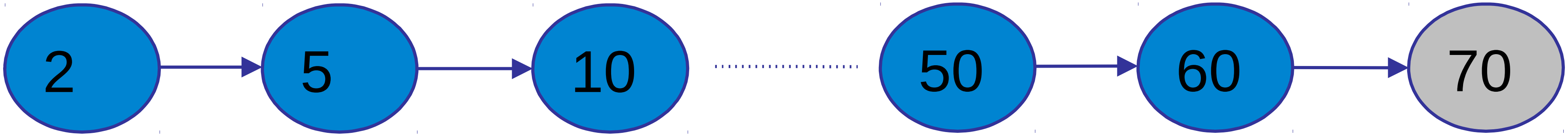
# Example

- Lazy Linked list (Insert “55”)



# Example

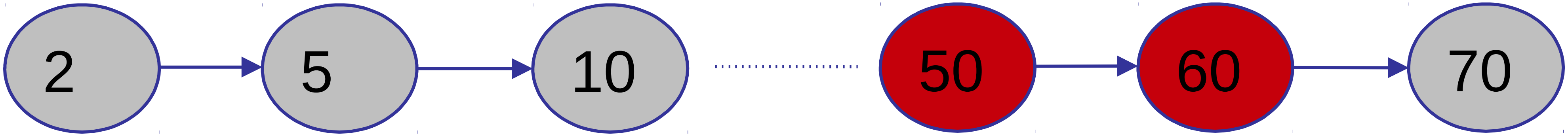
- Lazy Linked list (Insert “55”)



- Traversal (unmonitored)

# Example

- Lazy Linked list (Insert “55”)

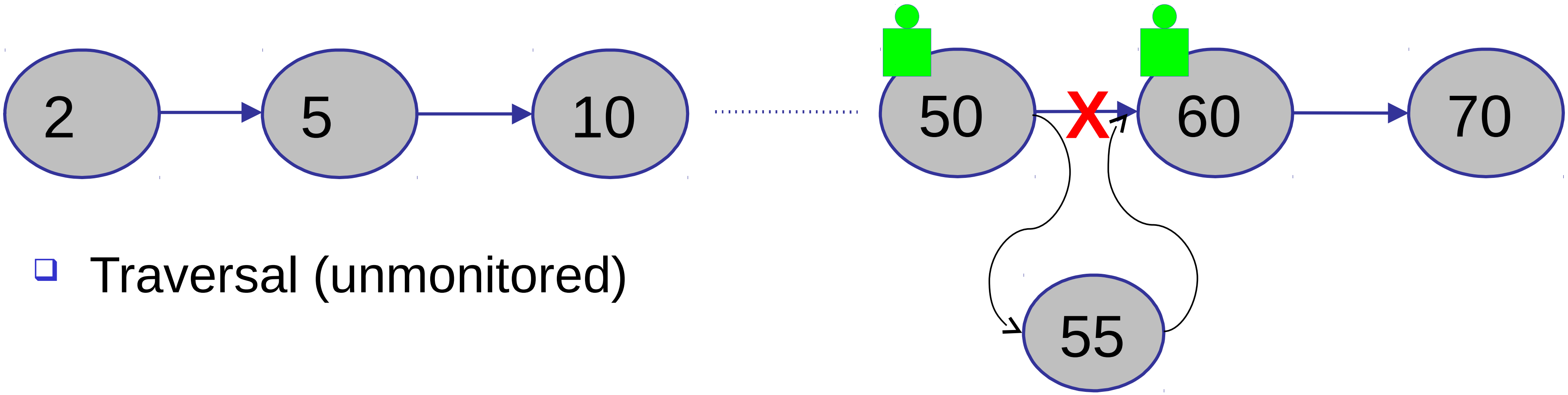


- Traversal (unmonitored)

- Validation

# Example

- Lazy Linked list (Insert “55”)



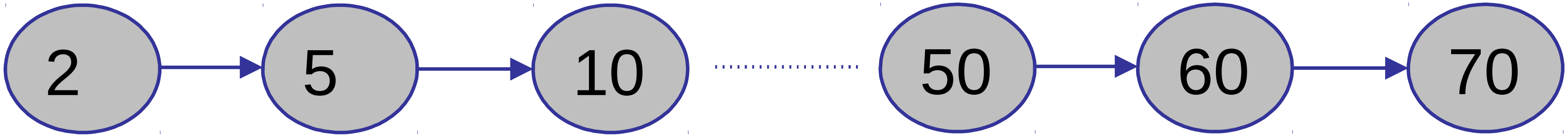
- Traversal (unmonitored)
- Validation
- Commit

# To Make it Transactional

- Results of traversal are saved in local objects:
  - Semantic read-set: to be validated.
  - Semantic write-set: to be published at commit.

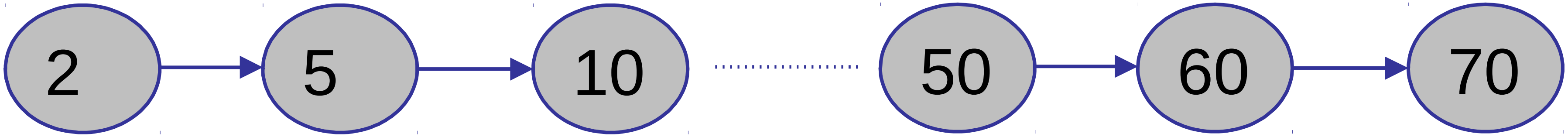
# To Make it Transactional

- Example: Linked list (Insert “55”)



# To Make it Transactional

- Example: Linked list (Insert “55”)



read-set entry

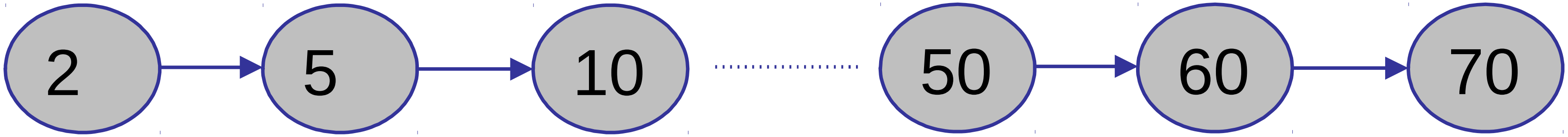
- Pred:50, curr:60

write-set entry

- Pred:50, curr:60, new:55

# To Make it Transactional

- Example: Linked list (Insert “55”)



read-set entry

- Pred:50, curr:60

write-set entry

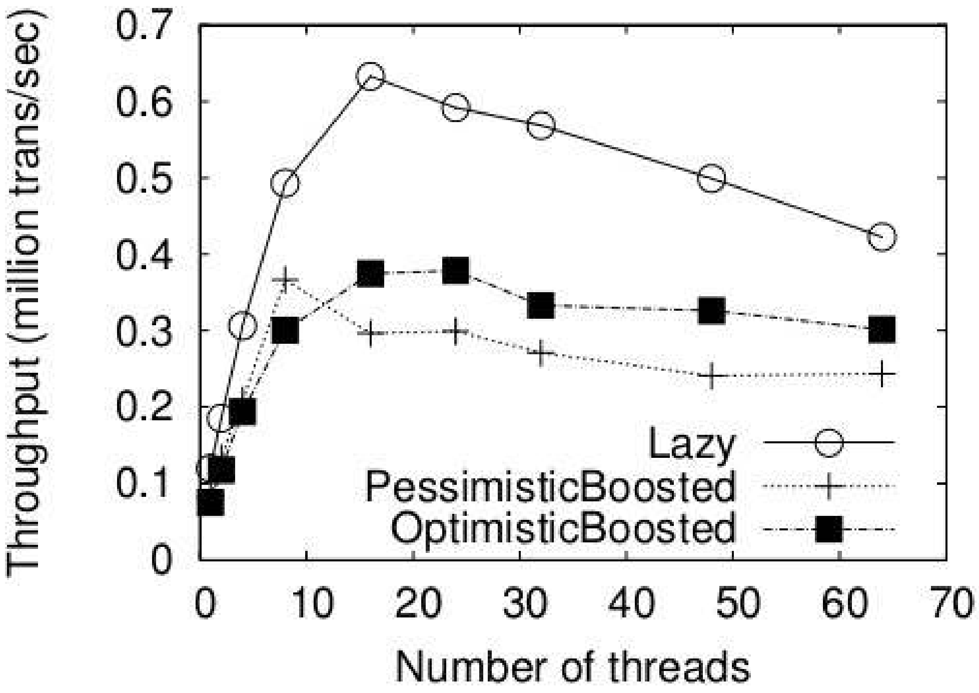
- Pred:50, curr:60, new:55

- Guidelines to guarantee opacity (see OPODIS'14 paper)

# Specific Optimizations

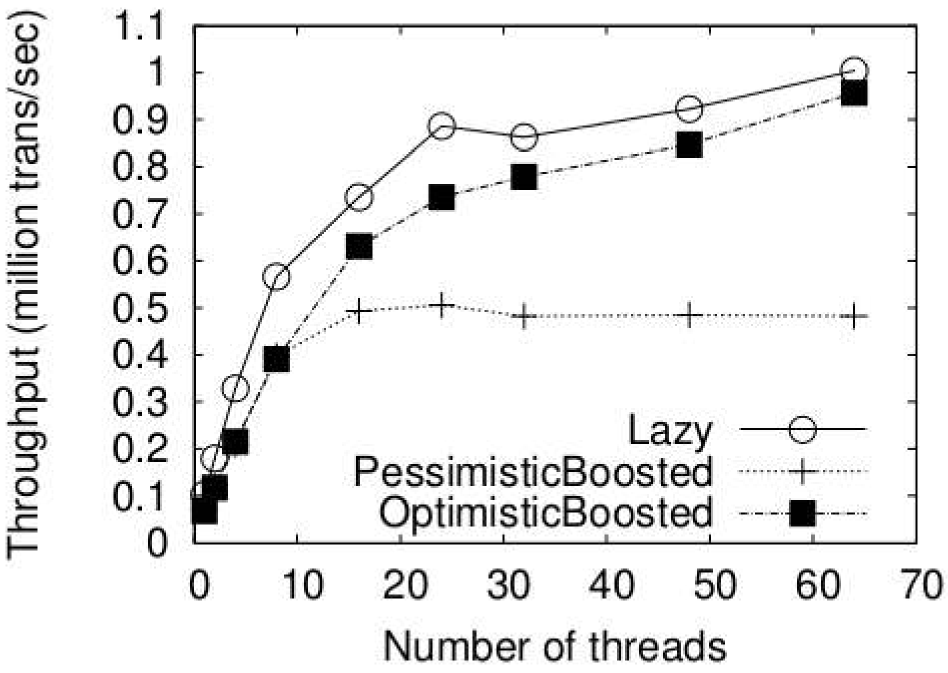
- Example optimizations on Linked-List and Skip-List
  - Local elimination:
    - Ex. Add(x) then Remove(x).
    - No need to access the shared data structure.

## Results



Skip-list 512 Nodes

5 ops/transaction



Skip-list 64K Nodes

5 ops/transaction

# Transactional Interference-less Balanced Tree

# Transactional Interference-less Balanced Trees

- Transactional: Functionality (following OTB's G1, G2).

- Interference-less: Performance (following OTB's G3).

# The Next Question

- Which concurrent balanced tree design fits OTB?

# The Next Question

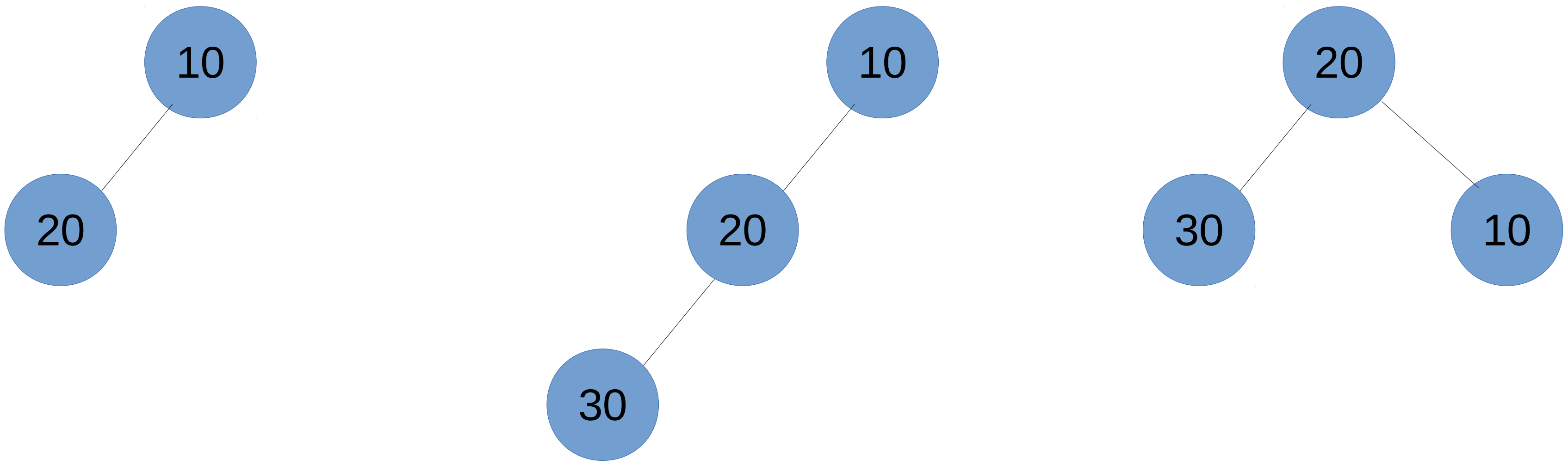
- Which concurrent balanced tree design fits OTB?

**Contention-Friendly Tree**

**Crain, Gramoli, & Raynal'13**

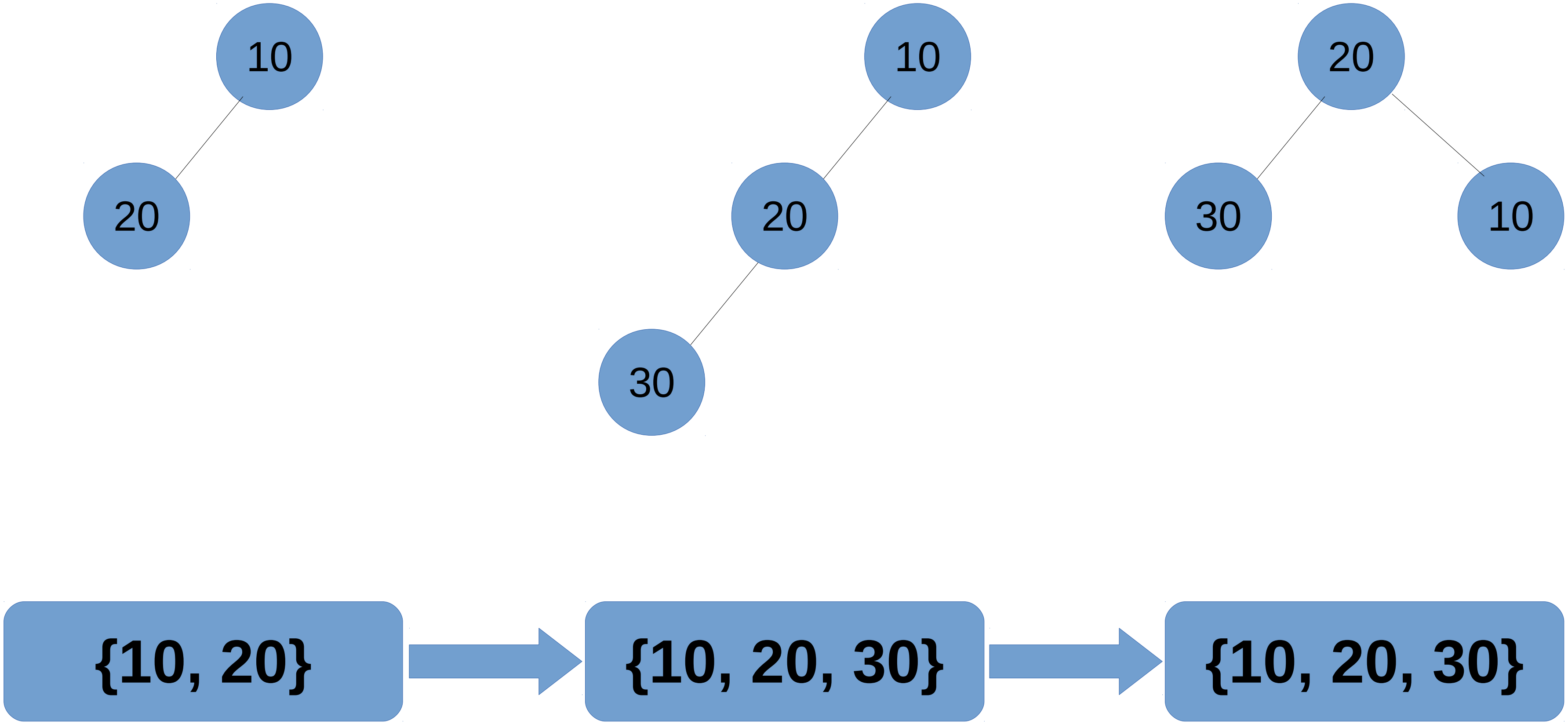
# CF-Tree

- Example: Insert 30.



# CF-Tree

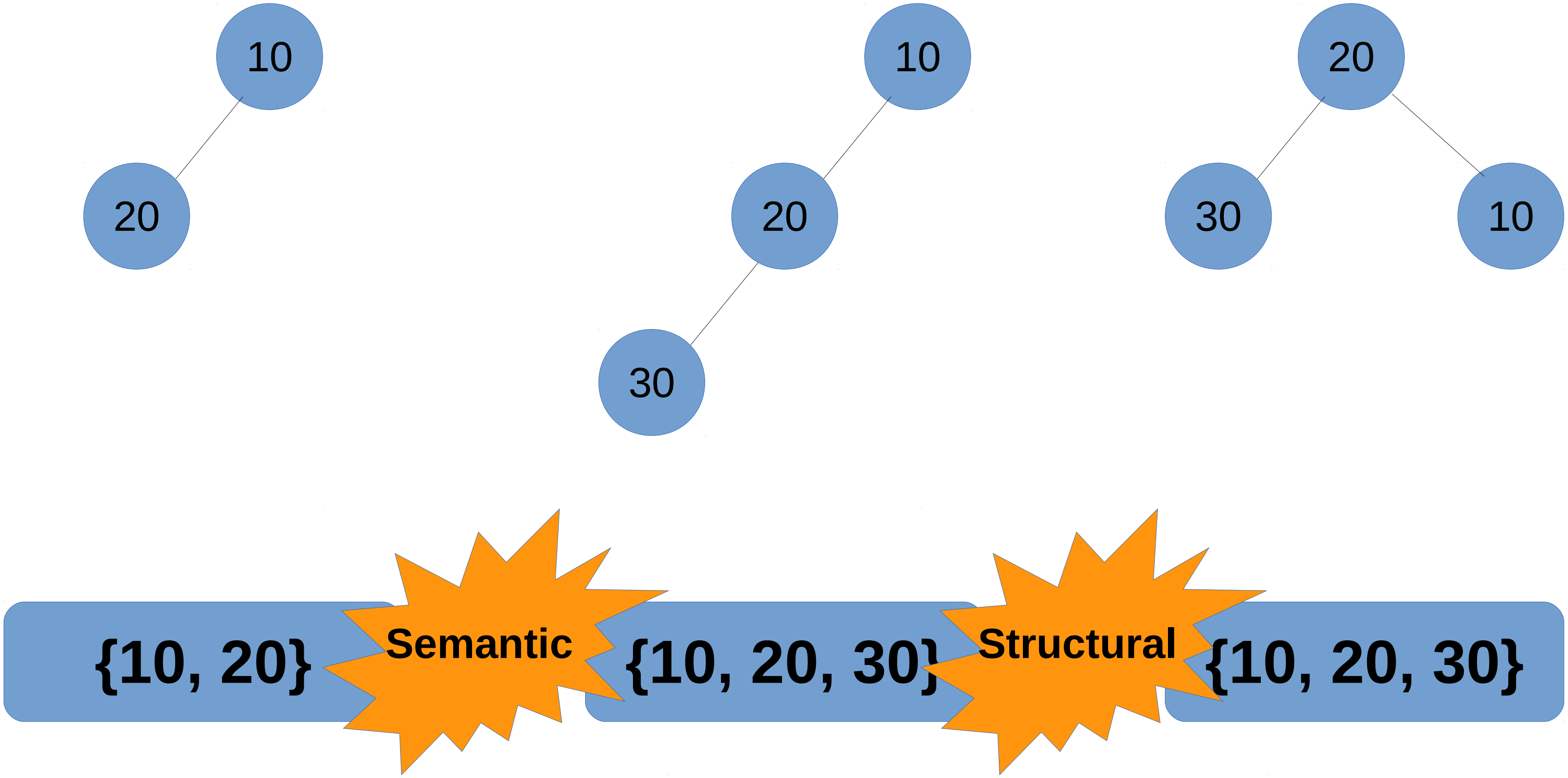
- Example: Insert 30.



{10, 20} → {10, 20, 30} → {10, 20, 30}

# CF-Tree

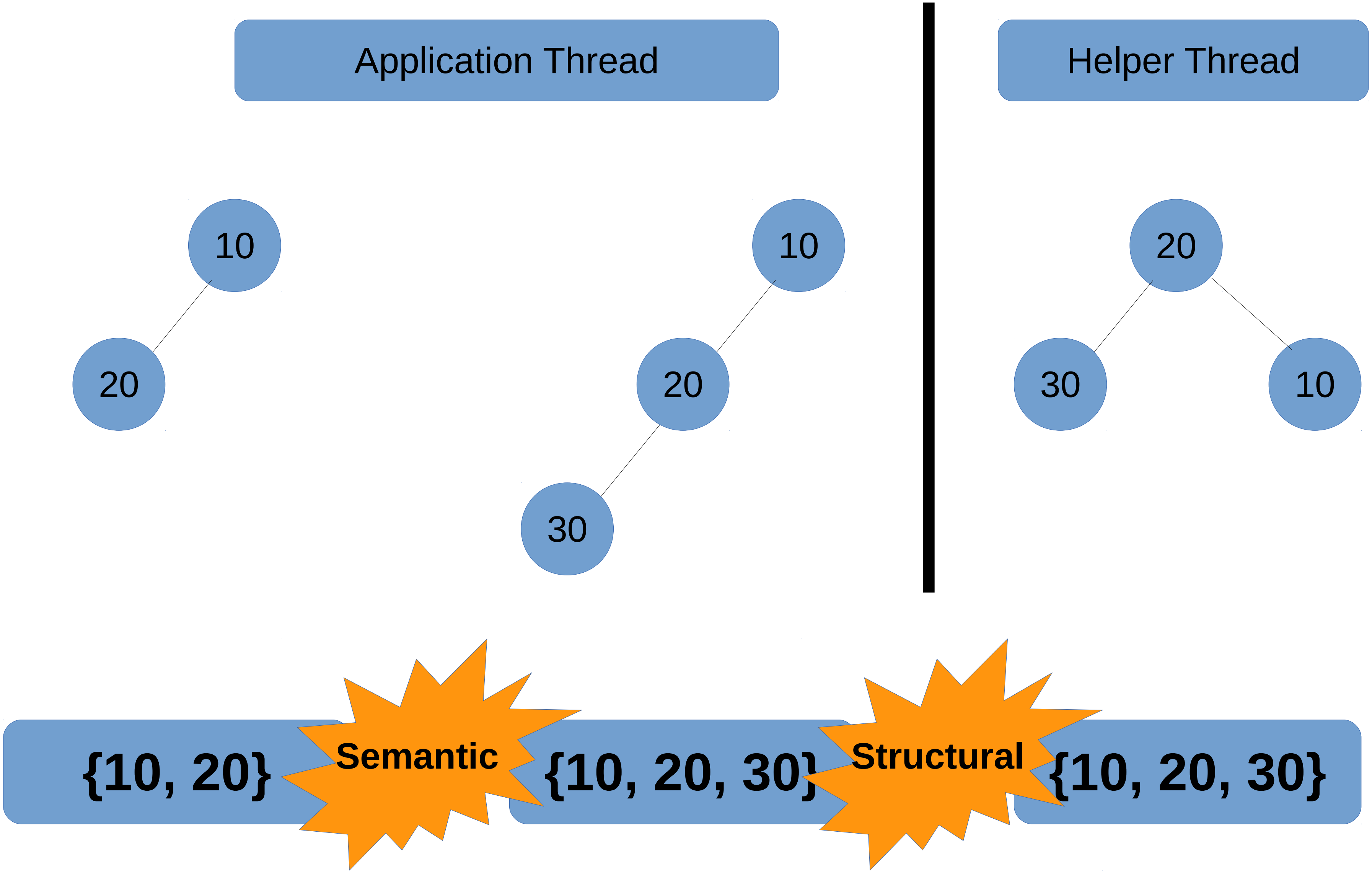
- Example: Insert 30.



{10, 20} Semantic {10, 20, 30} Structural {10, 20, 30}

# CF-Tree

- Example: Insert 30.



Application Thread

Helper Thread

{10, 20}

Semantic

{10, 20, 30}

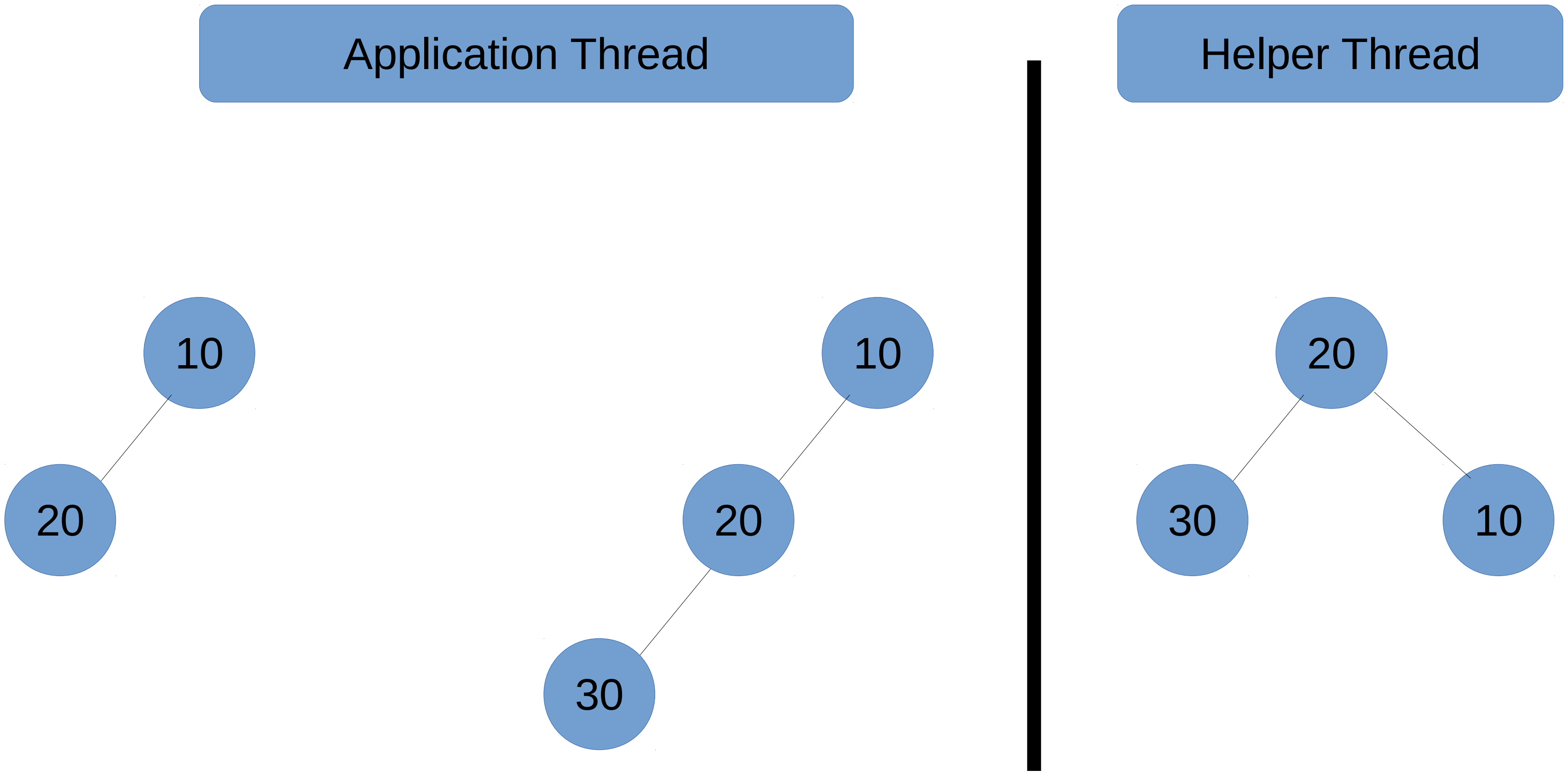
Structural

{10, 20, 30}

# Our Proposal

Transactionalizing CF-Tree using OTB
(TxCF-Tree)

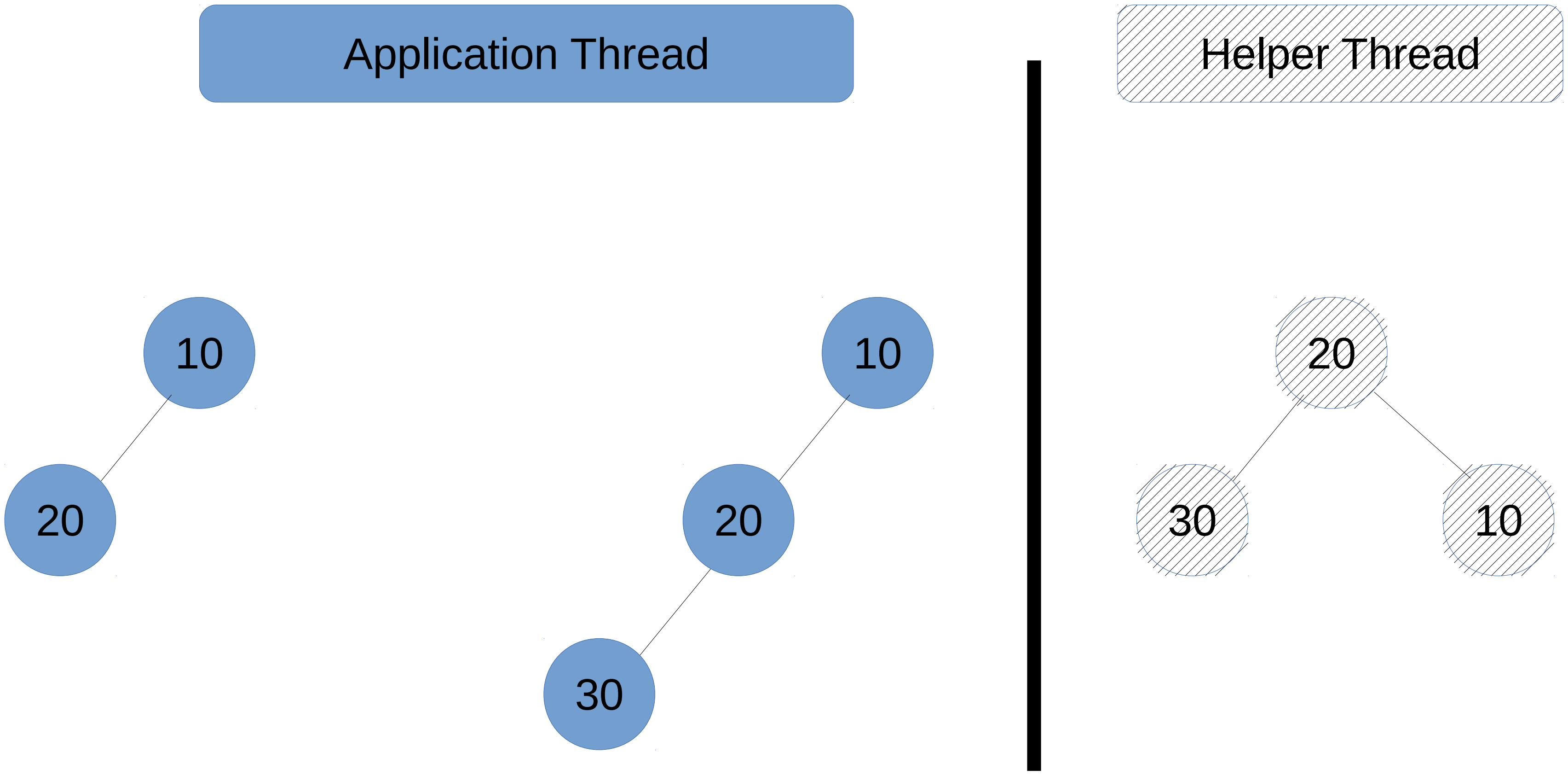
# TxCF-Tree



Application Thread

Helper Thread

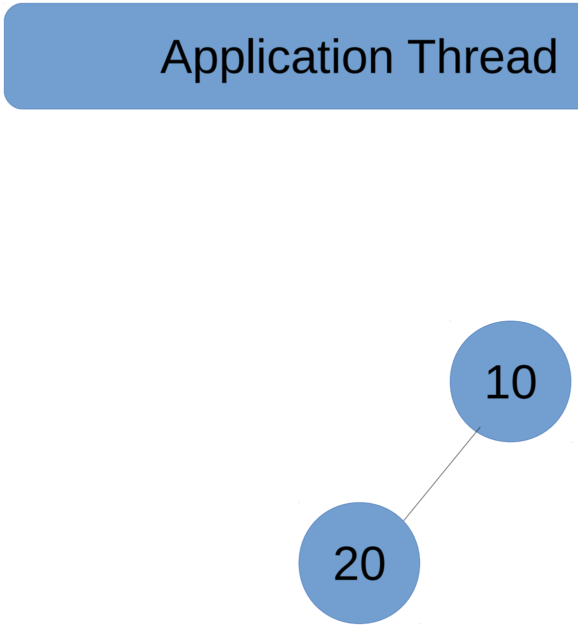
# TxCF-Tree



Application Thread

Helper Thread

# TxCF-Tree

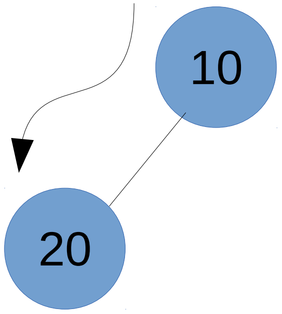


Application Thread

# TxCF-Tree

Application Thread

unmonitored traversal

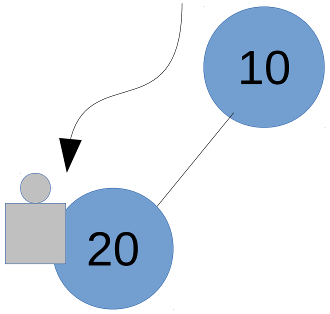


# TxCF-Tree

Application Thread

unmonitored traversal

Lock & Validate



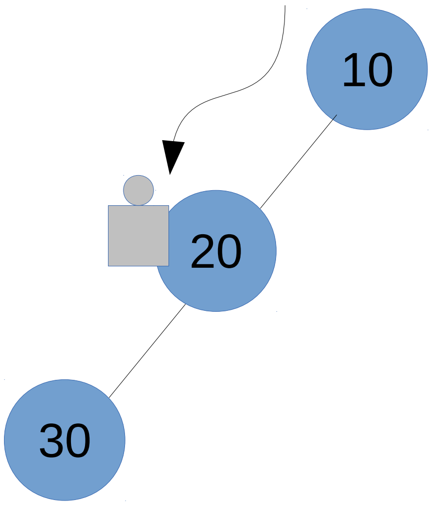
# TxCF-Tree

Application Thread

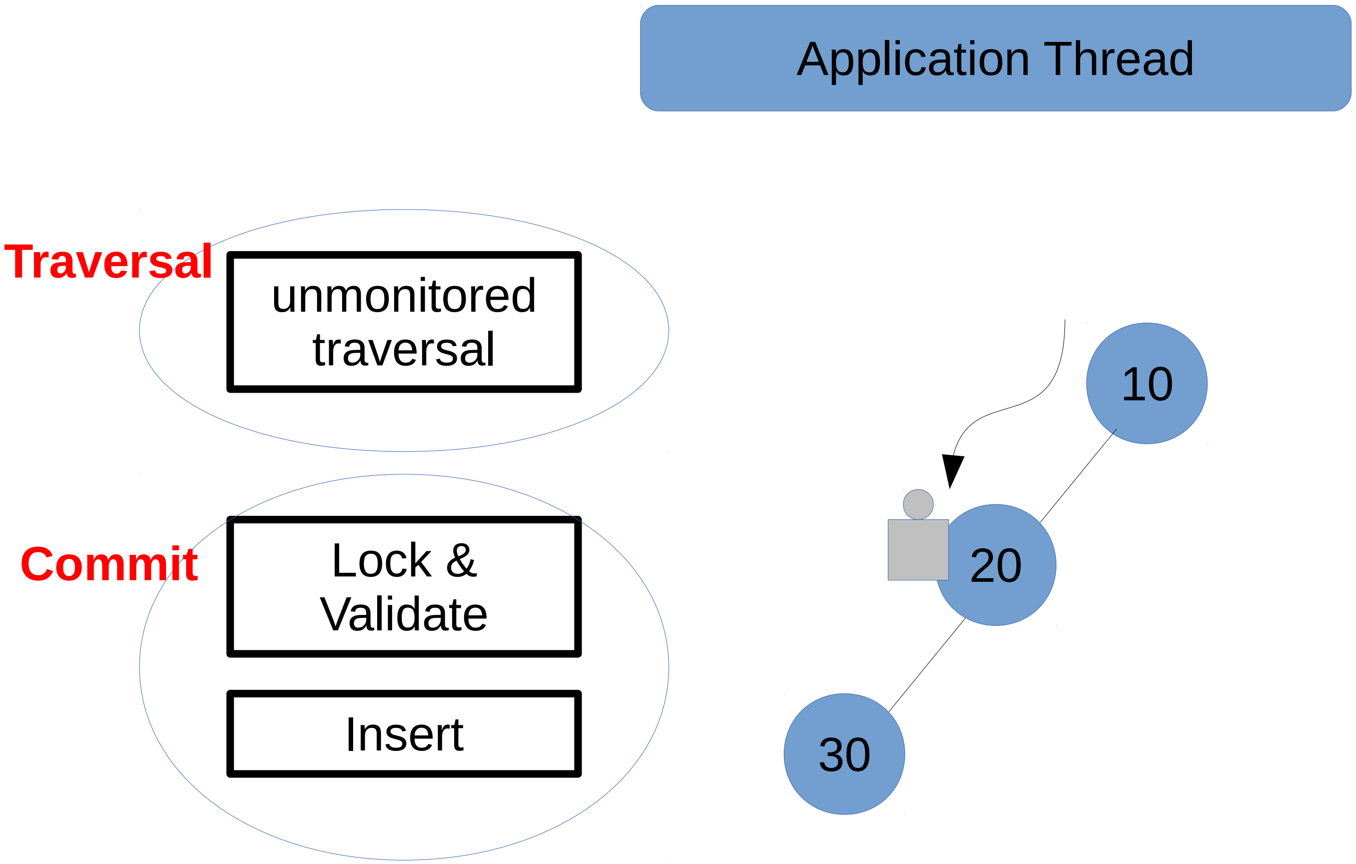
unmonitored traversal

Lock & Validate

Insert



# TxCF-Tree



Application Thread

Traversal

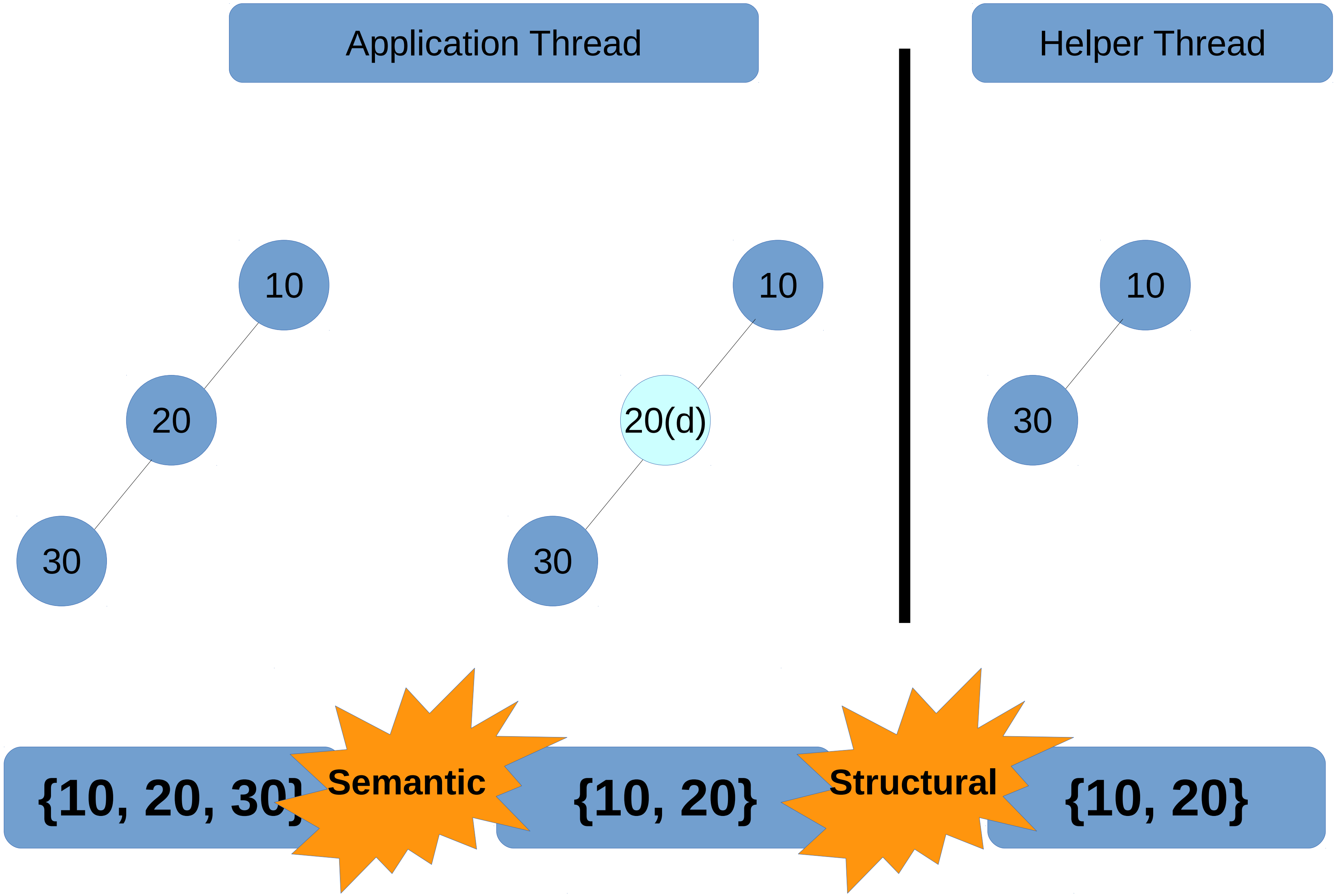
unmonitored traversal

Commit

Lock & Validate

Insert

# Remove is similar...



Application Thread

Helper Thread

{10, 20, 30}

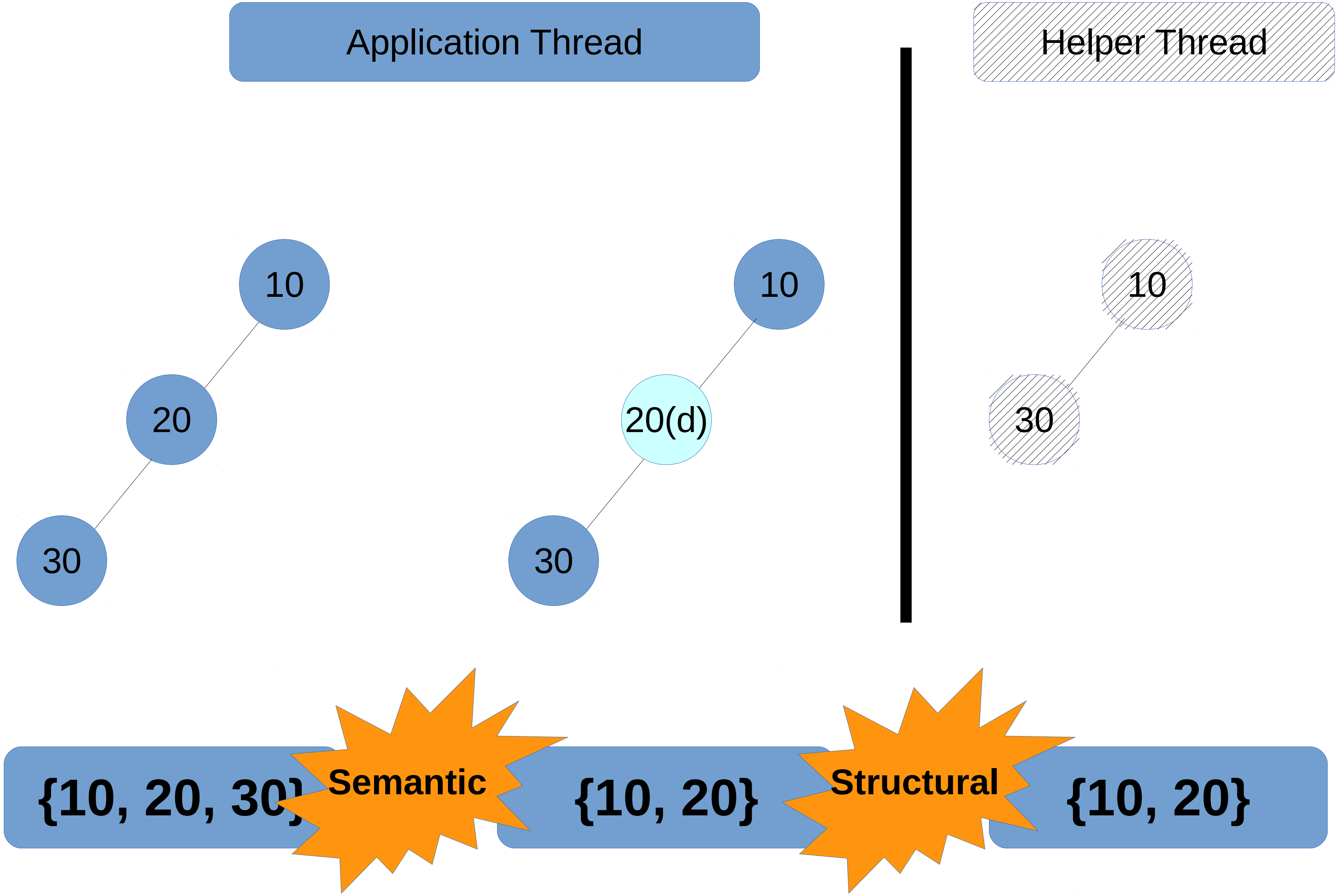
Semantic

{10, 20}

Structural

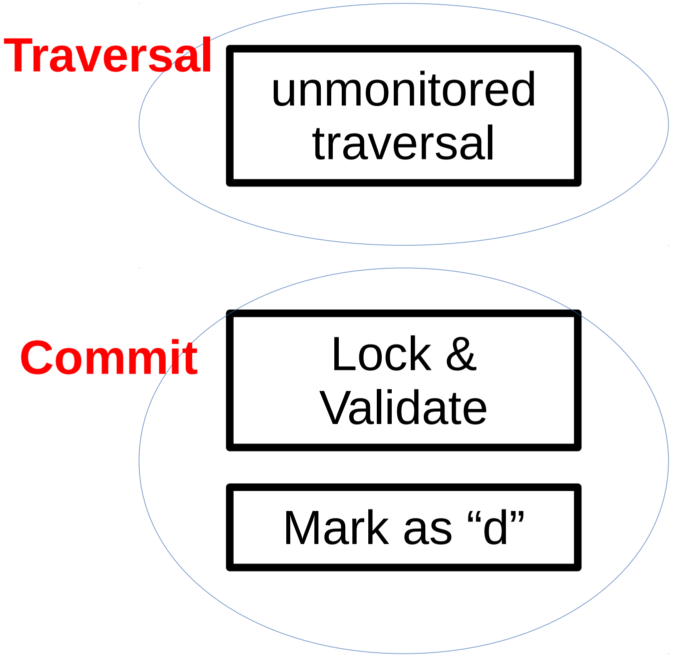
{10, 20}

# Remove is similar...



# Remove is similar...

Application Thread



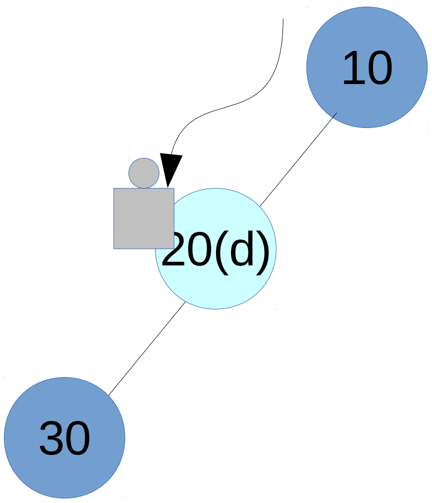
Traversal

unmonitored traversal

Commit

Lock & Validate

Mark as “d”



# Transactional Interference-less Tree

# Transactional Interference-less Tree

- How
  - Step 1: CF-Tree!!
  - Step 2: Always give the highest priority to semantic operations over structural operations.

# Transactional Interference-less Tree

- How
  - Step 1: CF-Tree!!
  - Step 2: Always give the highest priority to semantic operations over structural operations.

- Why
  - Aborting transactions rolls back all its operations (including the non-conflicting ones).
  - Long transactions are more prone to interfere with the helper thread.

# Two building blocks

- Structural Locks.

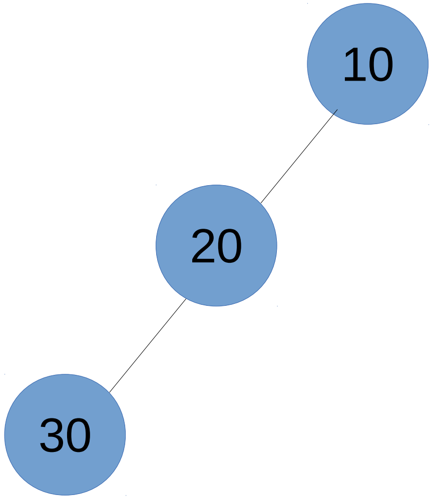
- Structural Invalidation.

# Structural Locks

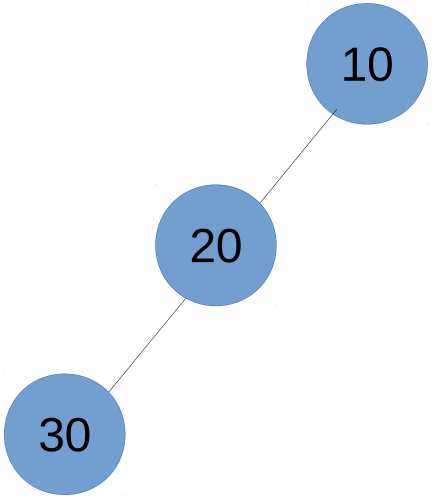
# Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**



**A concurrent delete(30)**

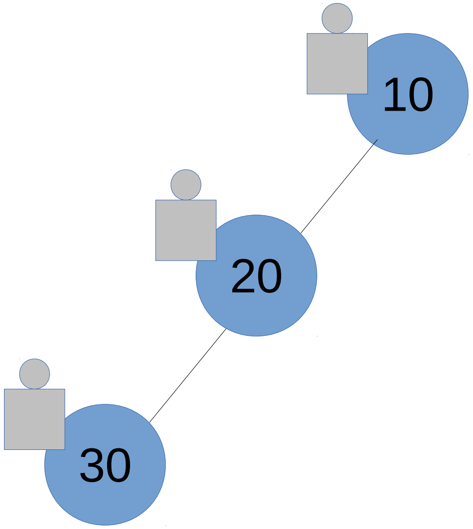


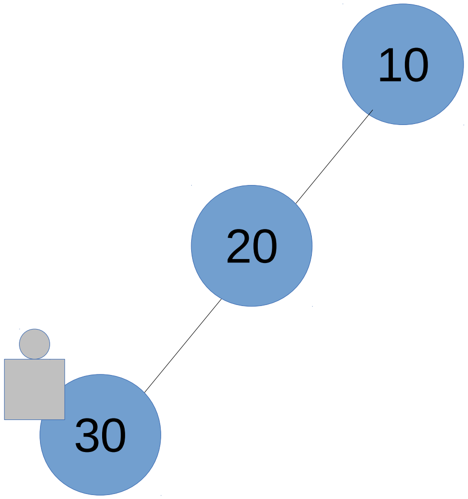
# Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**

**A concurrent delete(30)**

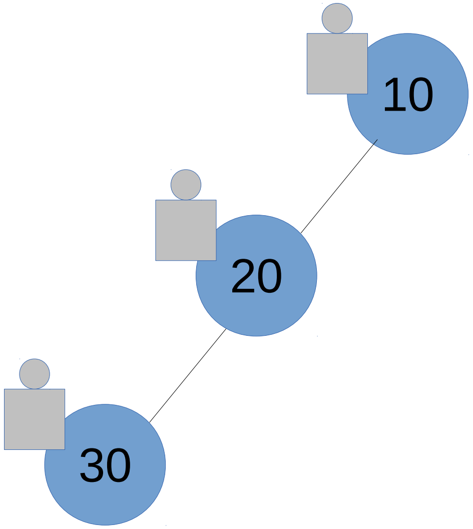




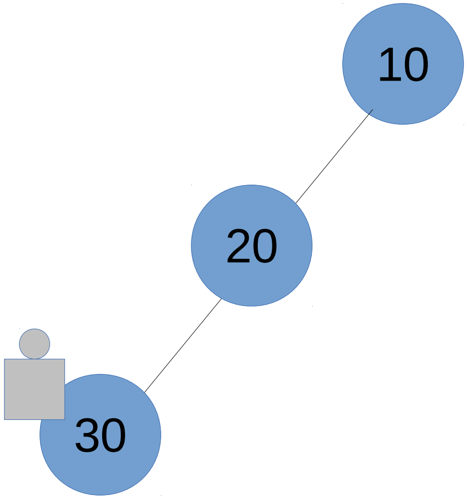
# Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**



**A concurrent delete(30)**

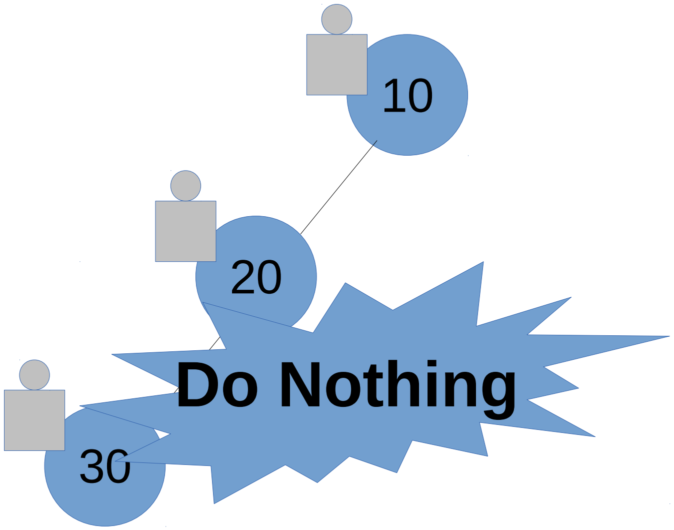


**T1 observes that “30” is locked**
**What is the best to do in both cases?**

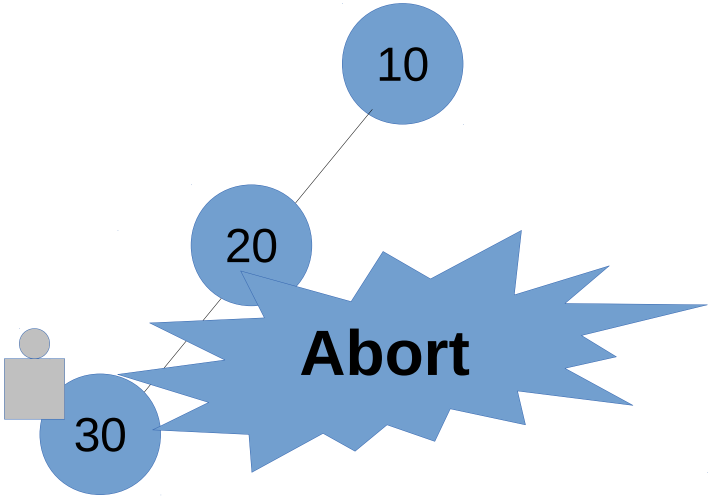
# Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**



**A concurrent delete(30)**



**T1 observes that “30” is locked**

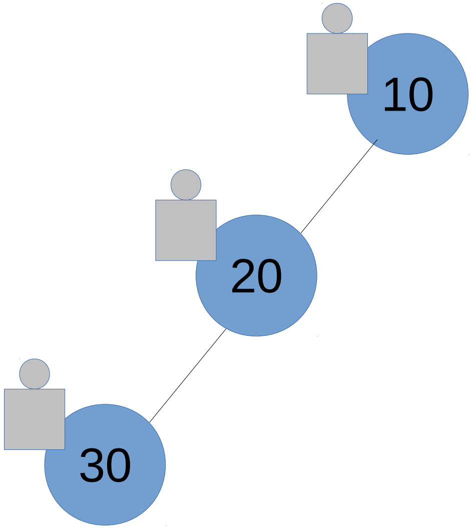
**What is the best to do in both cases?**

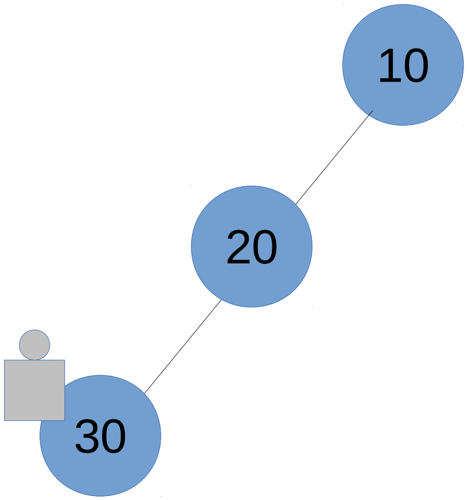
# Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**

**A concurrent delete(30)**



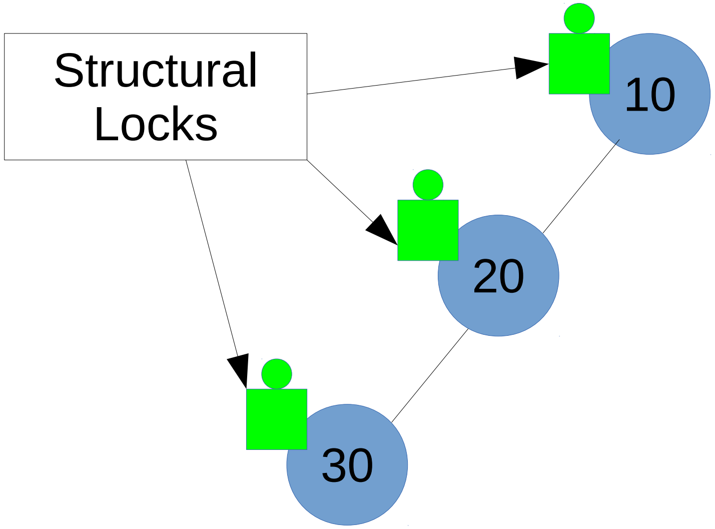


**Solution?**

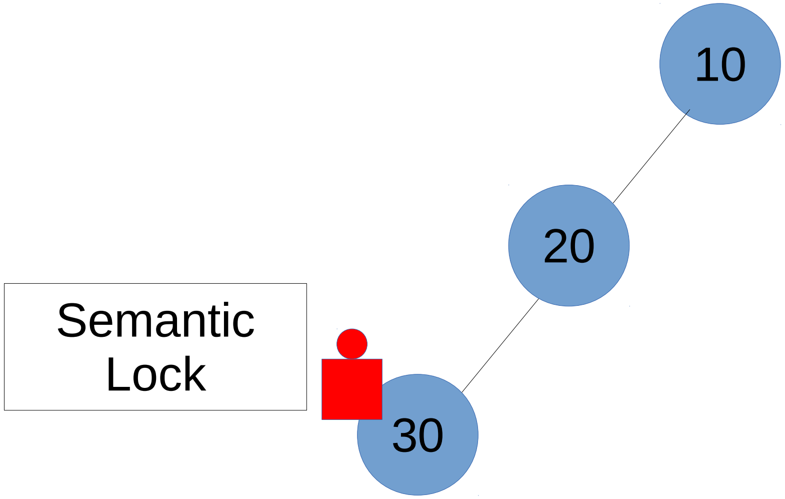
# Structural Locks

- Transaction T1 wants to delete 30.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**



**A concurrent delete(30)**



**Solution?**

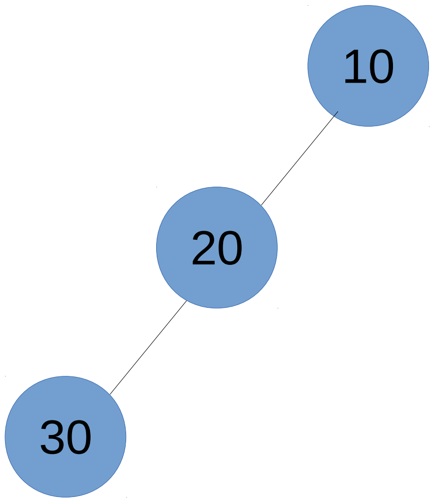
**Two types of locks**

# Structural Invalidation

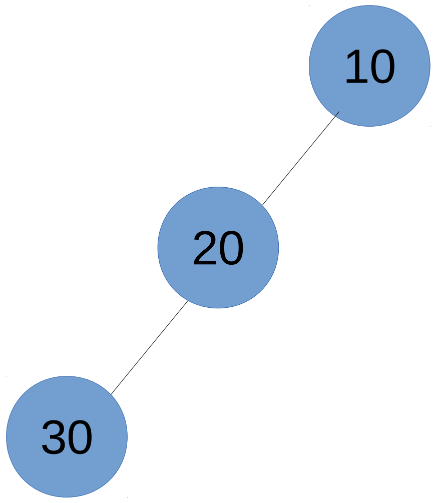
# Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**



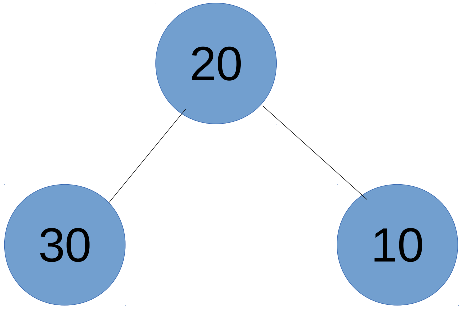
**A concurrent insert(15)**



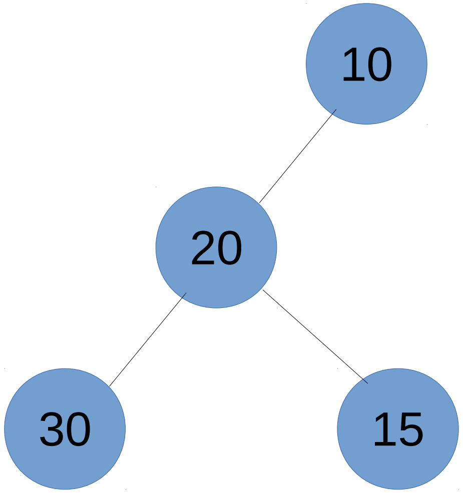
# Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

A concurrent rotation



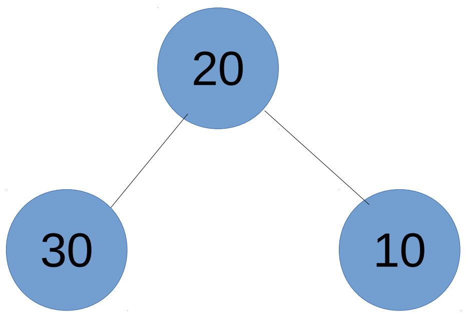
A concurrent insert(15)



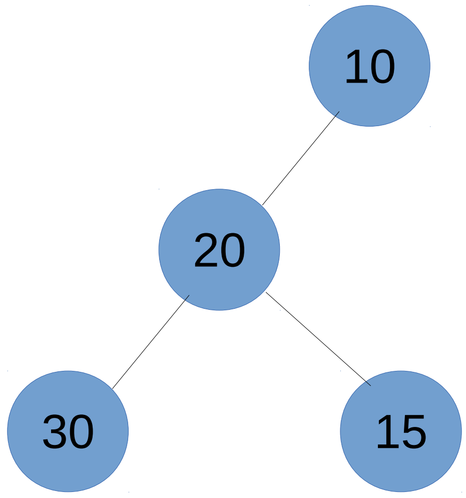
# Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**



**A concurrent insert(15)**



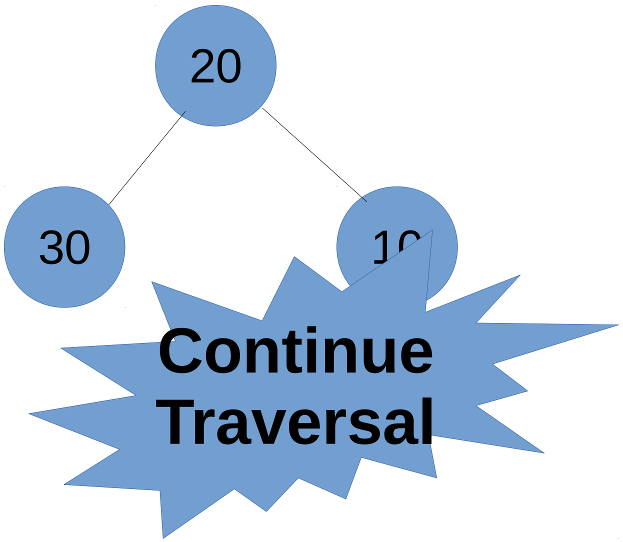
**T1 observes that the right child of “20” is not NULL**
**What is the best to do in both cases?**

# Structural Invalidation

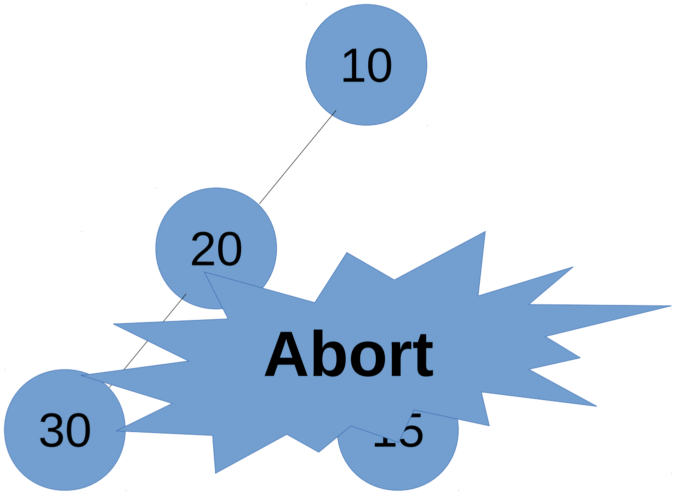
- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**

**A concurrent insert(15)**



**Continue Traversal**



**Abort**

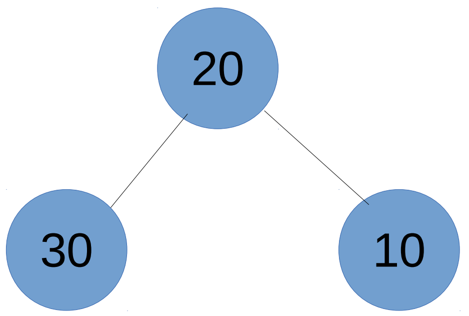
**T1 observes that the right child of "20" is not NULL**
**What is the best to do in both cases?**

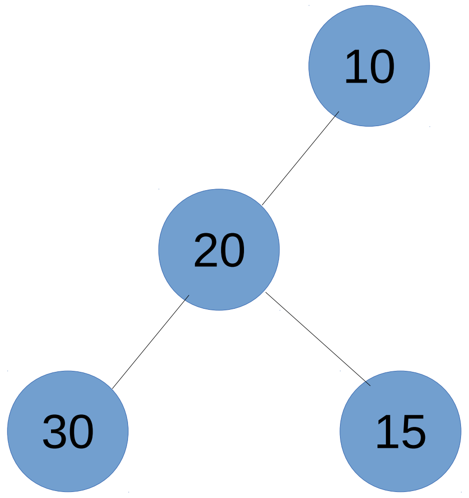
# Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**

**A concurrent insert(15)**



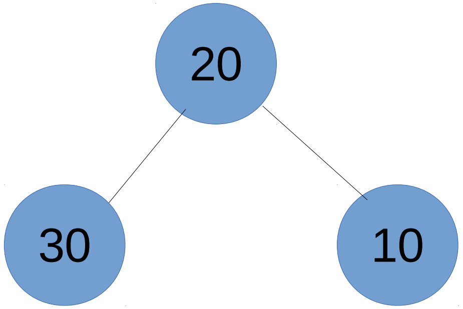


**Solution?**

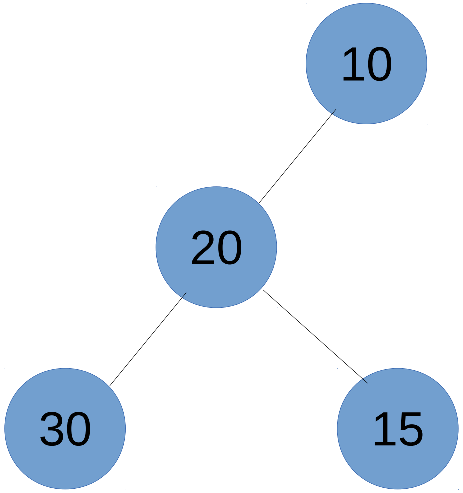
# Structural Invalidation

- Transaction T1 wants to insert 15.
- after traversal and before commit, assume 2 scenarios

**A concurrent rotation**



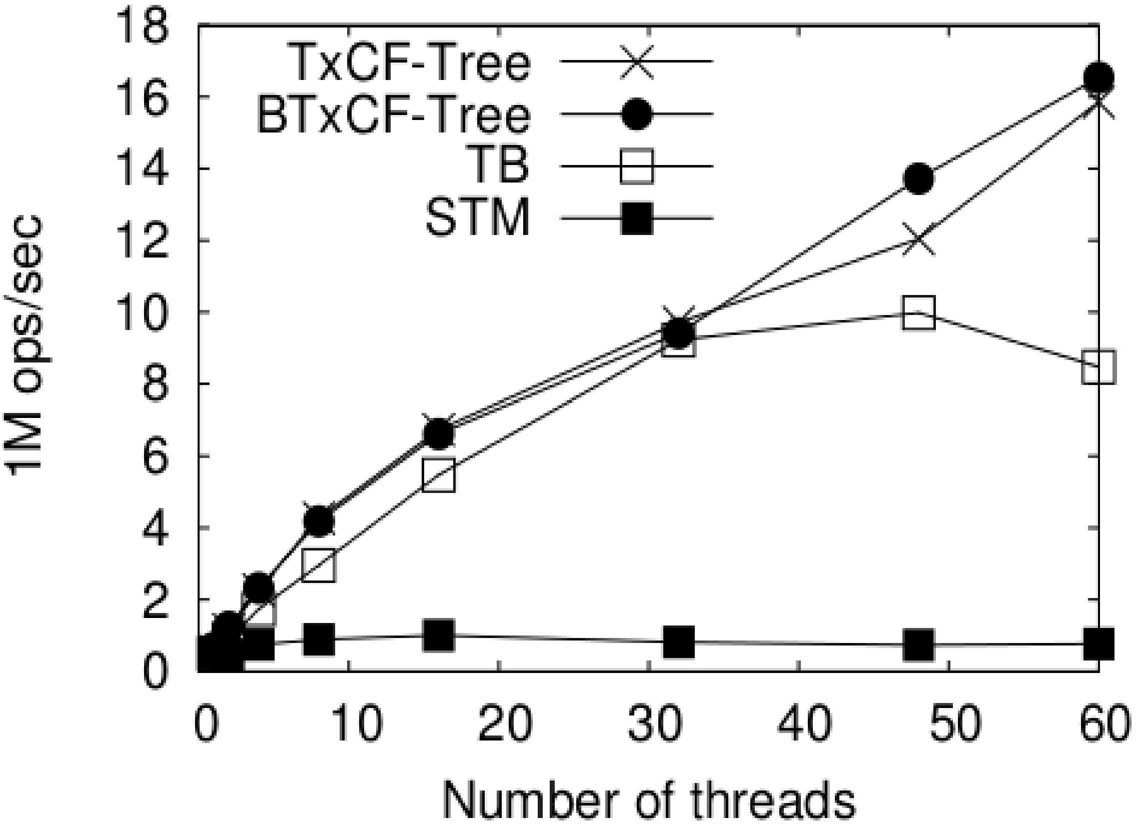
**A concurrent insert(15)**



**Solution?**

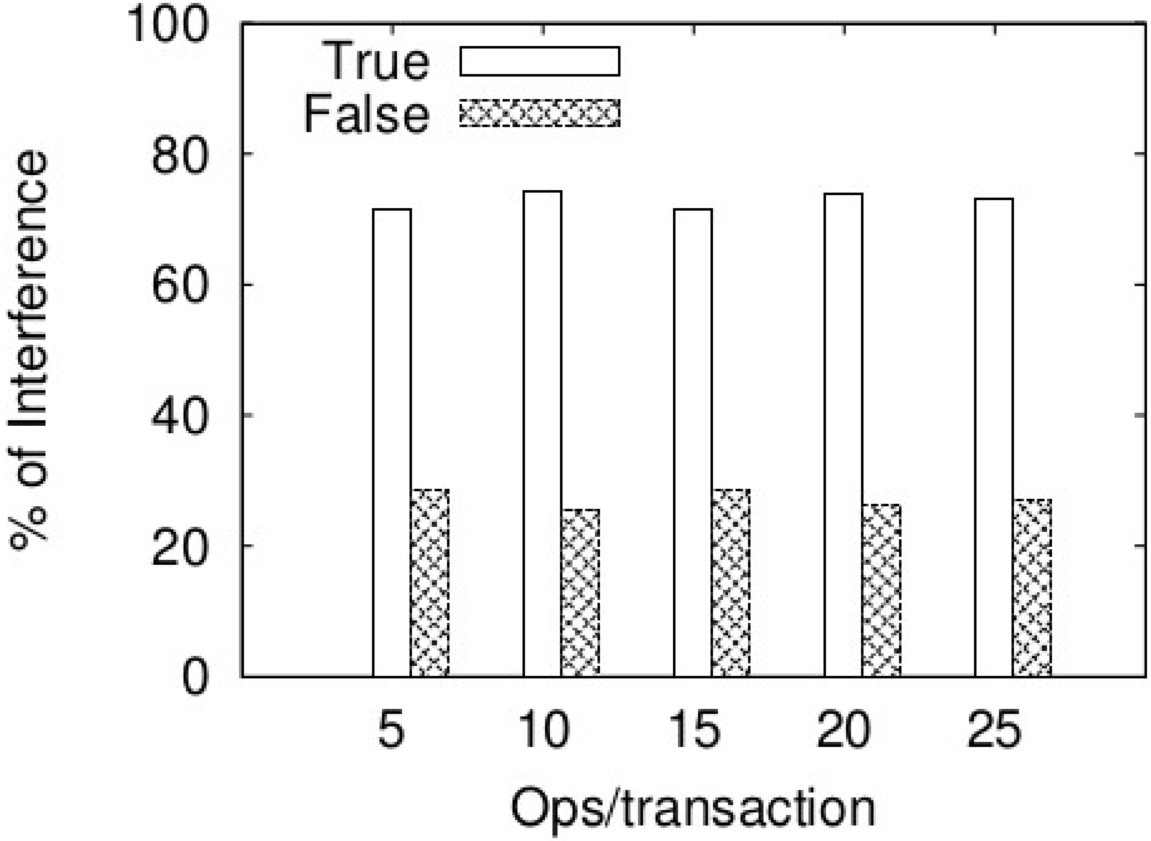
**Continue Traversal anyway**

# Evaluation



**AMD 64-cores, size 10K nodes, 50% reads, 5 ops/transaction**

# Evaluation



**AMD 64-cores, size: 10K nodes , 32 threads, 50% reads, 5 ops/transaction**

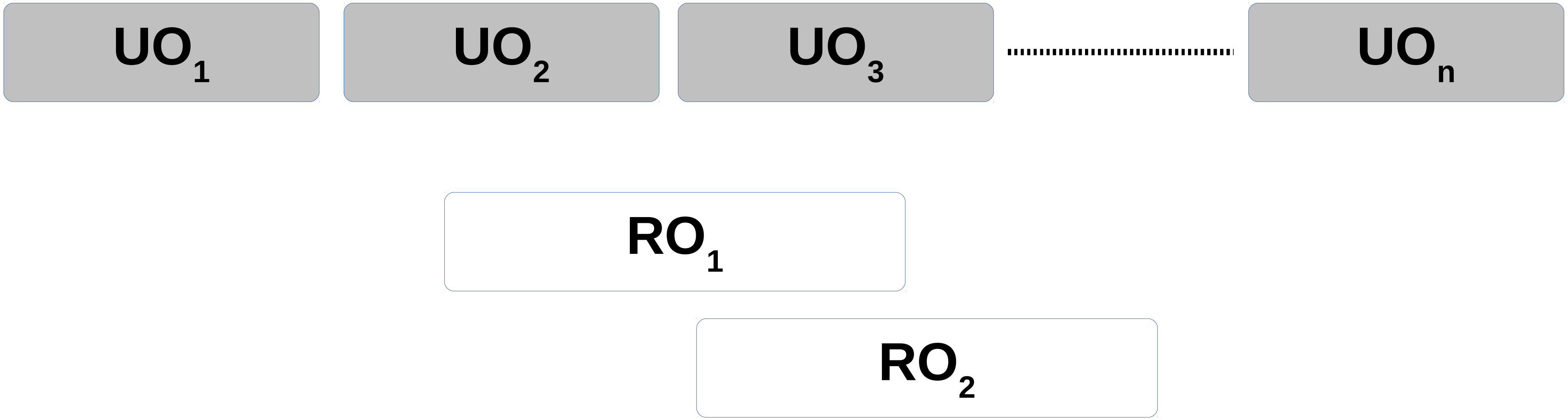
# Modeling Transactional Data Structures

## Concurrent Data Structures

- Different Designs and Implementations
  - Different ad-hoc approaches for proving correctness.

- Is there a unified model for concurrent data structures?
  - General enough.
  - Easy to use.

# SWMR Model (Lev-Ari et. Al, DISC'14)



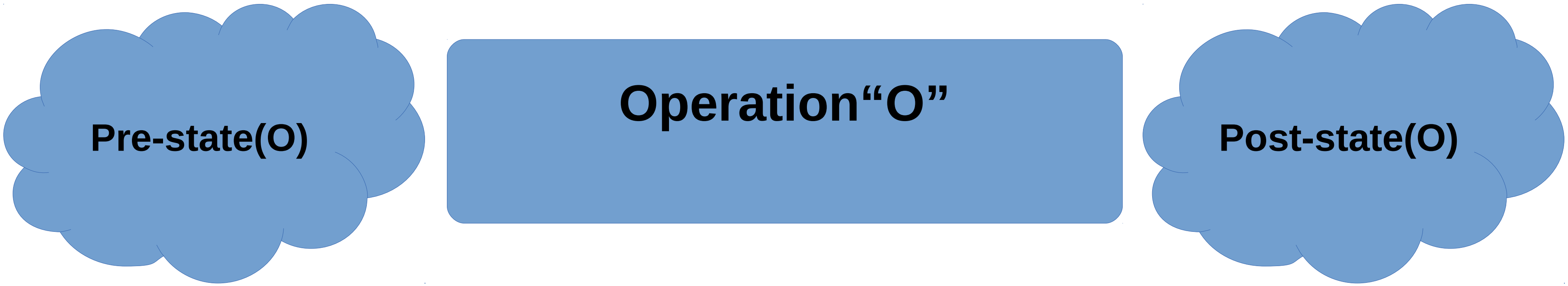
$UO_1$ $UO_2$ $UO_3$ $UO_n$

$RO_1$

$RO_2$

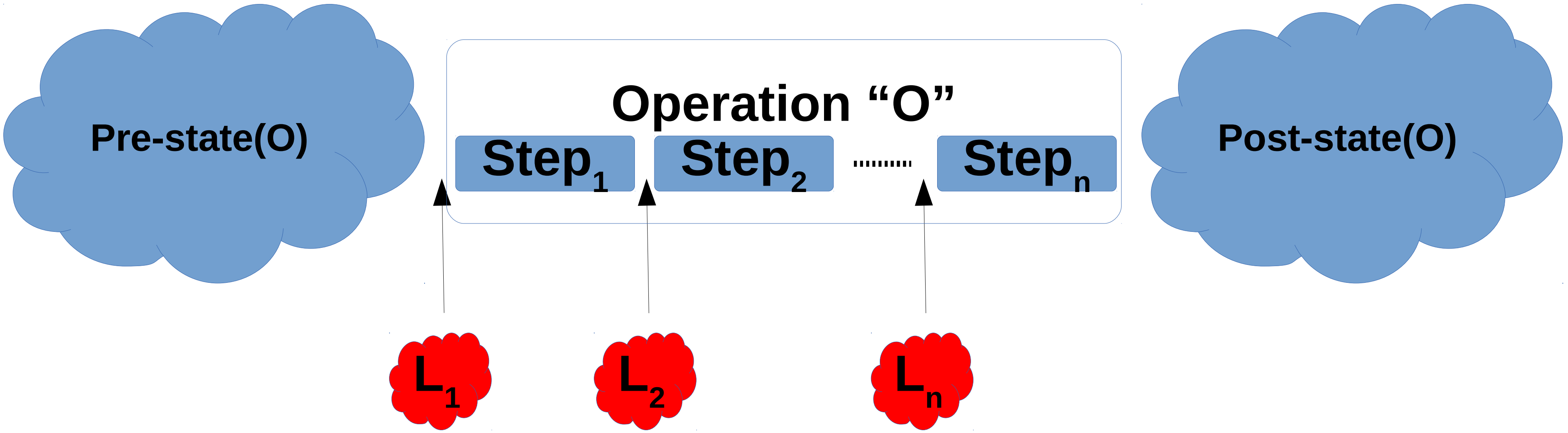
# Shared States

- Data Structure is represented as a set of shared variables.
- The values of those variables is the shared state of the data structure.

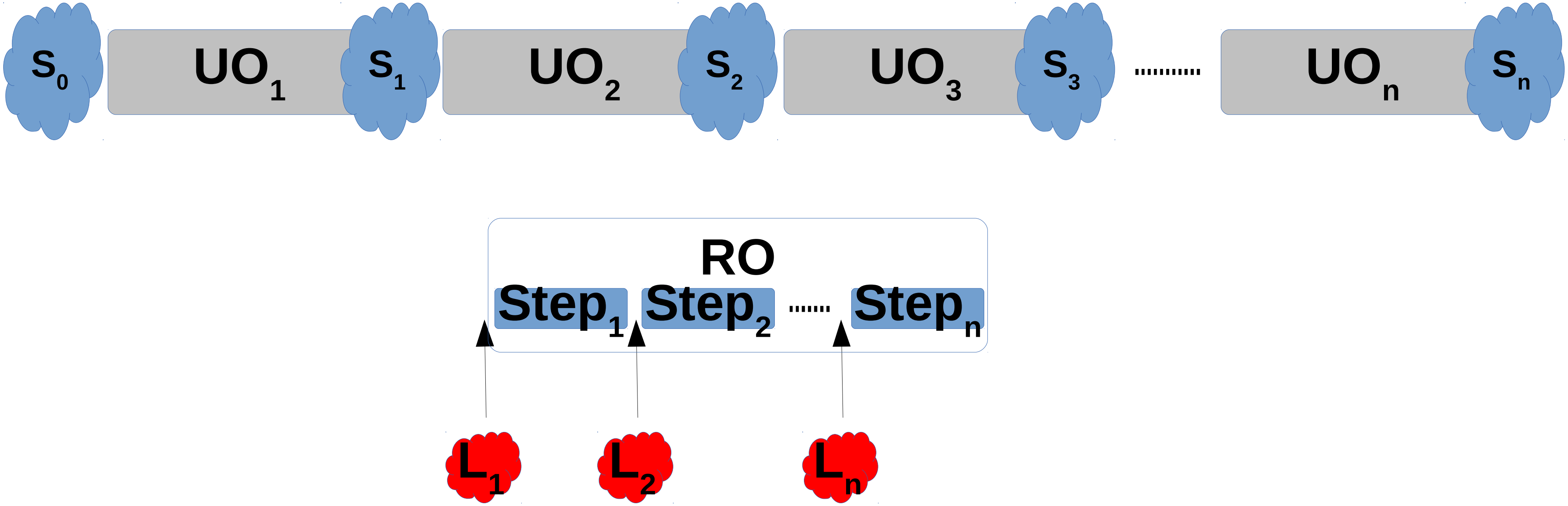


# Local States

- Operation is represented as a set of steps.
- The values of the operation's local variables before any step is the local state of the step.

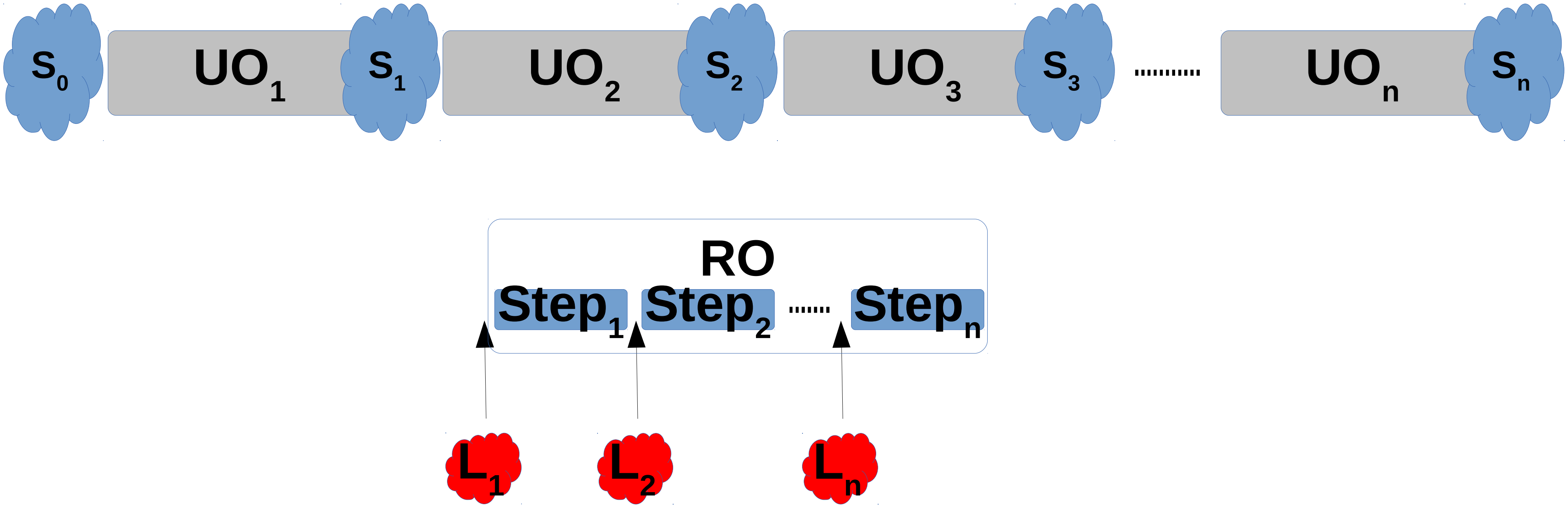


# SWMR Scenario

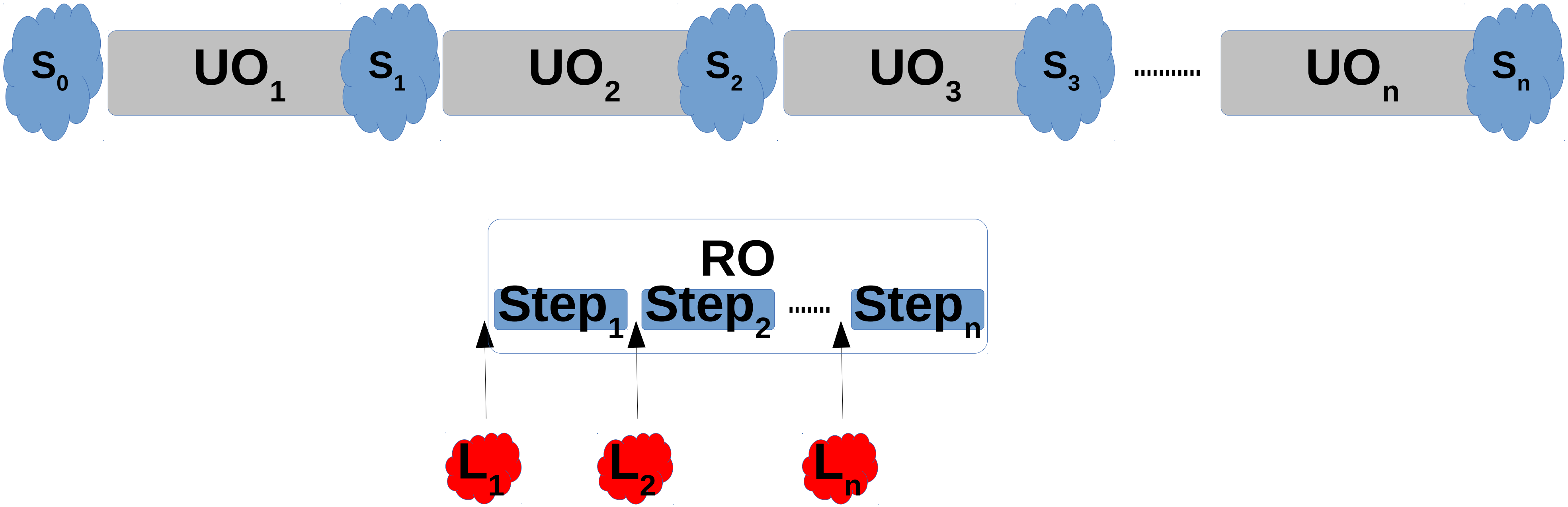


# Validity

# Validity

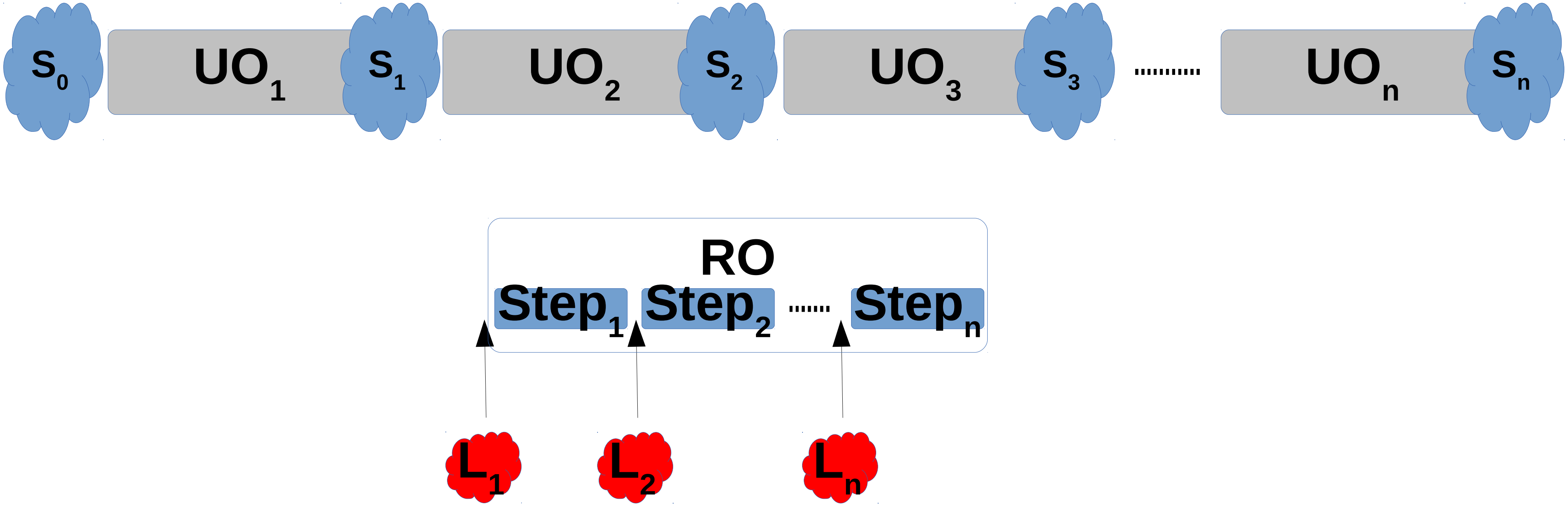


# Validity



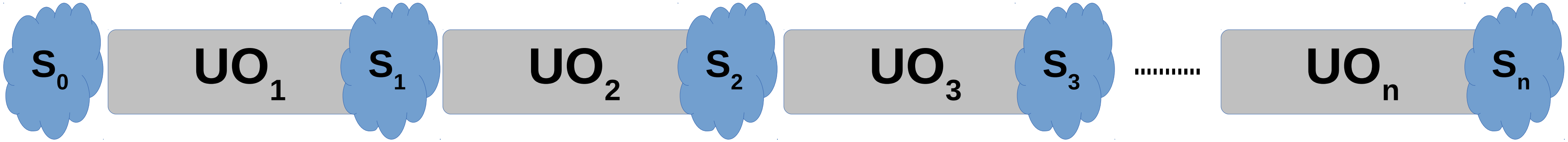
- All $S_i$ are sequentially reachable, so all $UO_i$ are valid.

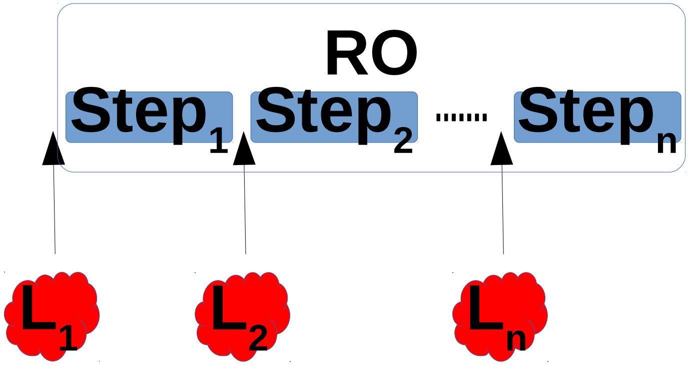
# Validity



- All $S_i$ are sequentially reachable, so all $UO_i$ are valid.
- $Step_i$ in RO is valid if there is $S_j$ such that a sequential execution of RO starting from $S_j$ reaches $L_i$.

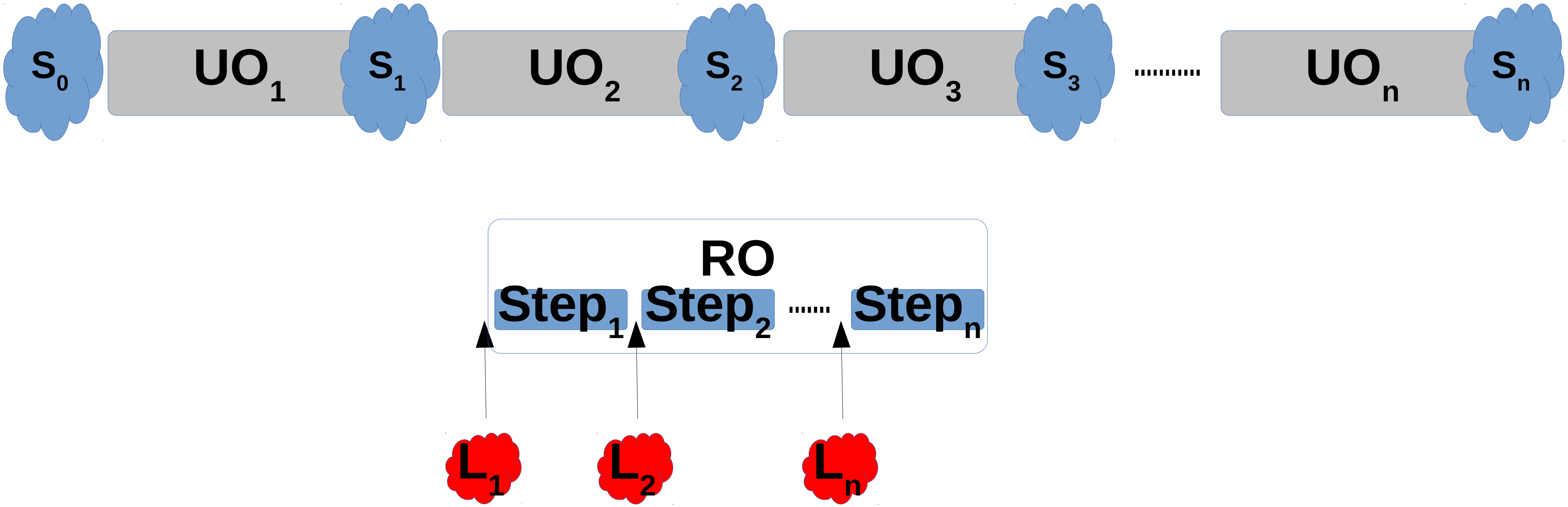
# Validity





- All $S_i$ are sequentially reachable, so all $UO_i$ are valid.
- $Step_j$ in RO is valid if there is $S_i$ such that a sequential execution of RO starting from $S_i$ reaches $L_j$.

# Validity



- All $S_i$ are sequentially reachable, so all $UO_i$ are valid.
- $Step_j$ in RO is valid if there is $S_i$ such that a sequential execution of RO starting from $S_i$ reaches $L_j$.

↓

- $Step_j$ in RO is valid if there is a “base point” where the “base condition” of $step_j$ holds.

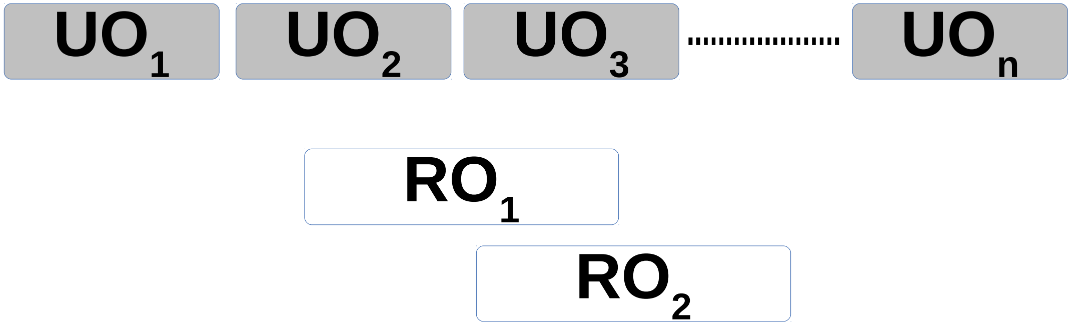
# Validity

- How to prove validity for any data structure.
  - Identify the base conditions for each step in each operation (it is sufficient to do so only for steps that access the shared memory).
  - Prove that in any concurrent execution, every step has a base point that satisfies its base condition.

# Validity

- How to prove validity for any data structure.
  - Identify the base conditions for each step in each operation (it is sufficient to do so only for steps that access the shared memory).
  - Prove that in any concurrent execution, every step has a base point that satisfies its base condition.

# Regularity

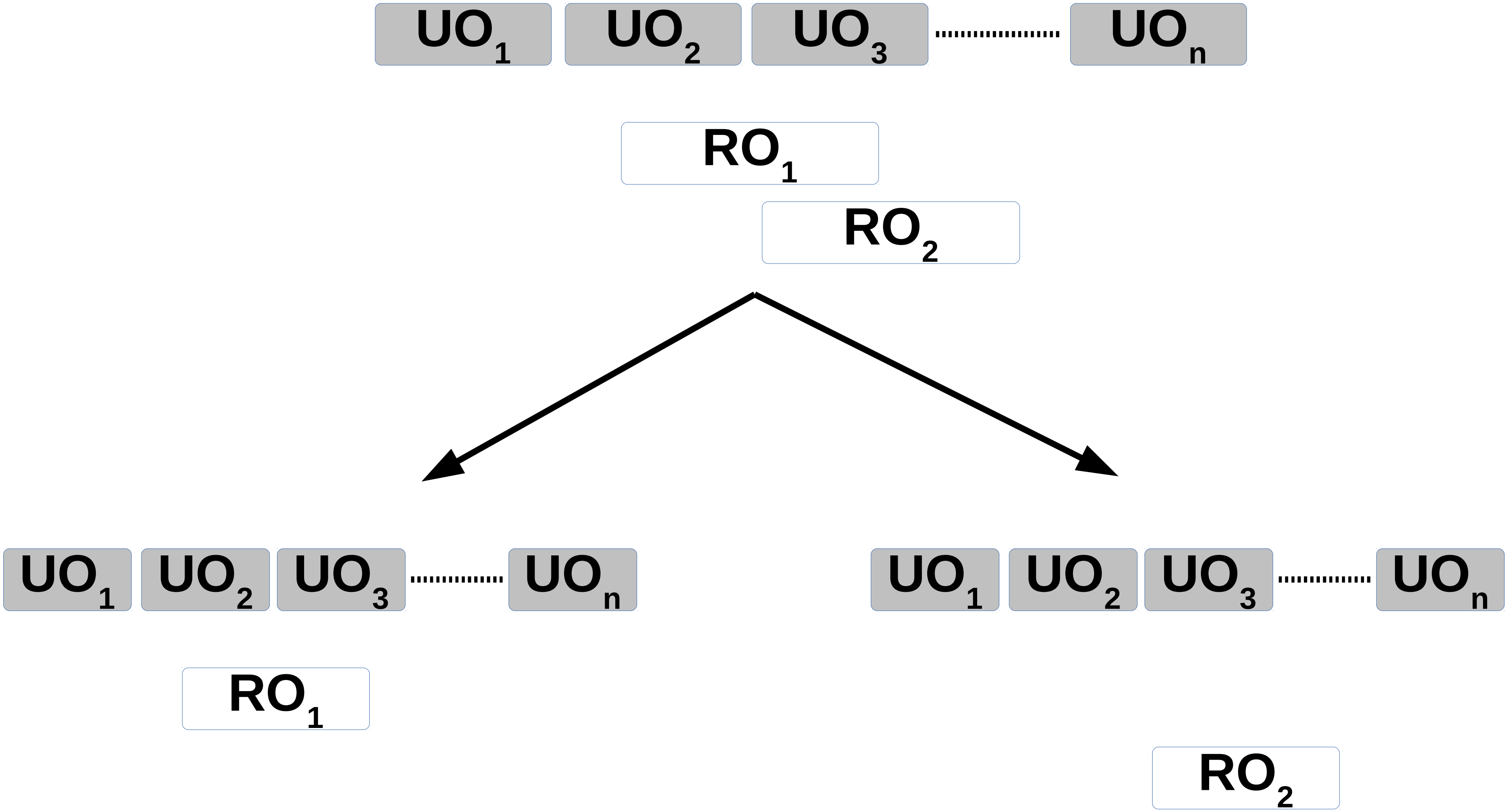


$UO_1$ $UO_2$ $UO_3$ .................... $UO_n$

$RO_1$

$RO_2$

# Regularity



$UO_1$ $UO_2$ $UO_3$ ........ $UO_n$

$RO_1$

$RO_2$

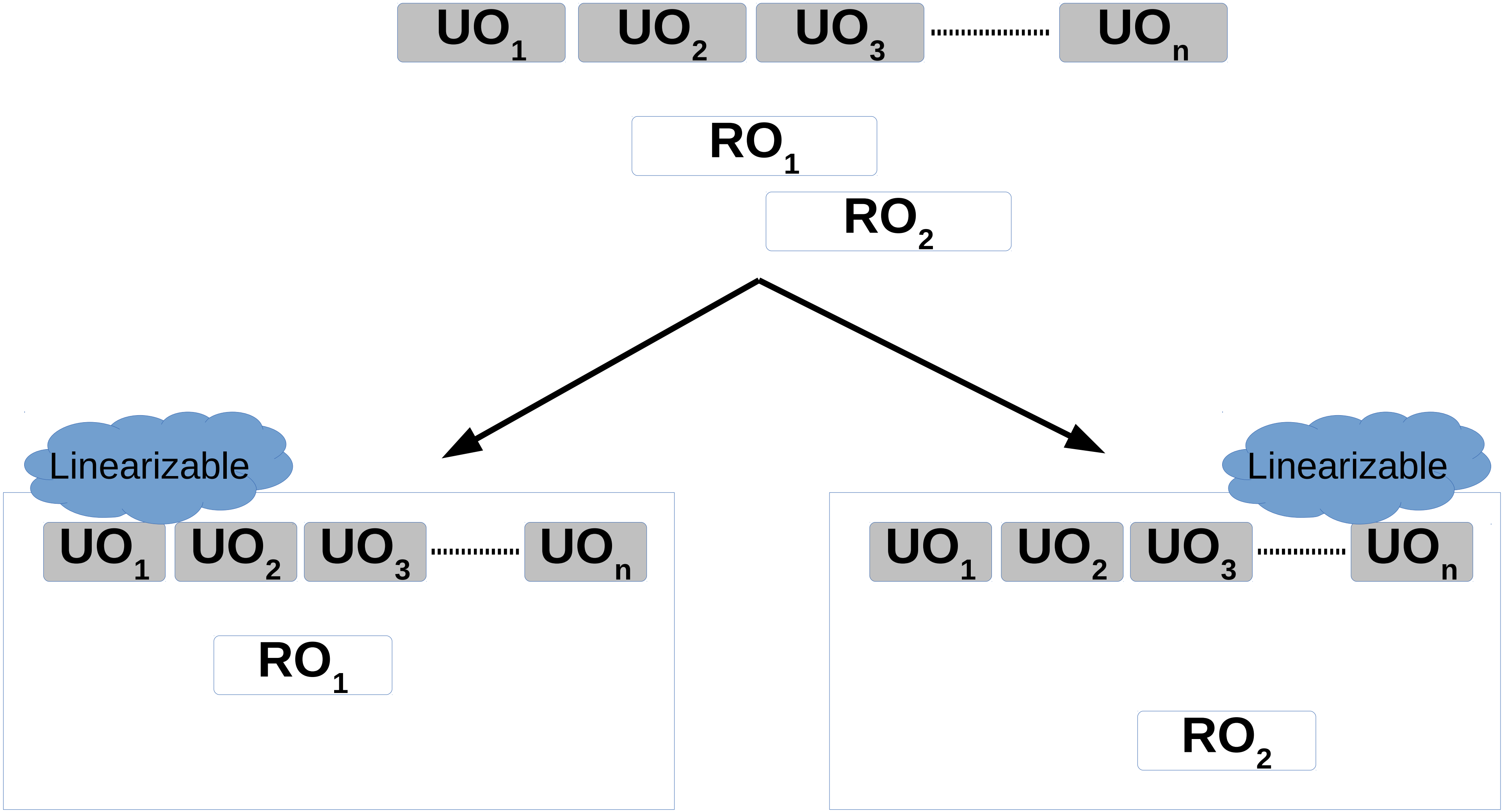
$UO_1$ $UO_2$ $UO_3$ ........ $UO_n$

$RO_1$

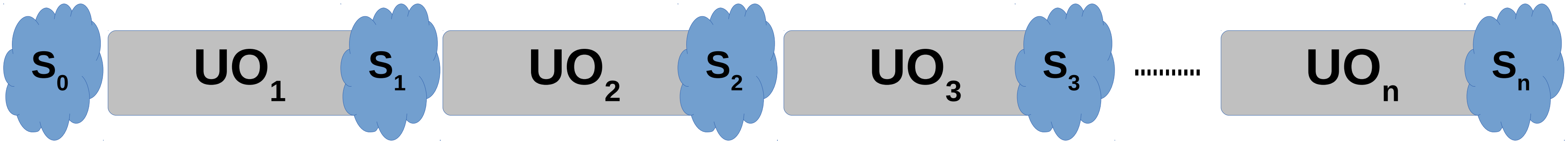
$UO_1$ $UO_2$ $UO_3$ ........ $UO_n$

$RO_2$

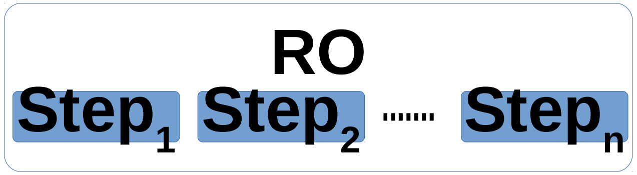
# Regularity


$UO_1$ $UO_2$ $UO_3$ $UO_n$
$RO_1$
$RO_2$
Linearizable
$UO_1$ $UO_2$ $UO_3$ $UO_n$
$RO_1$
Linearizable
$UO_1$ $UO_2$ $UO_3$ $UO_n$
$RO_2$

# Regularity



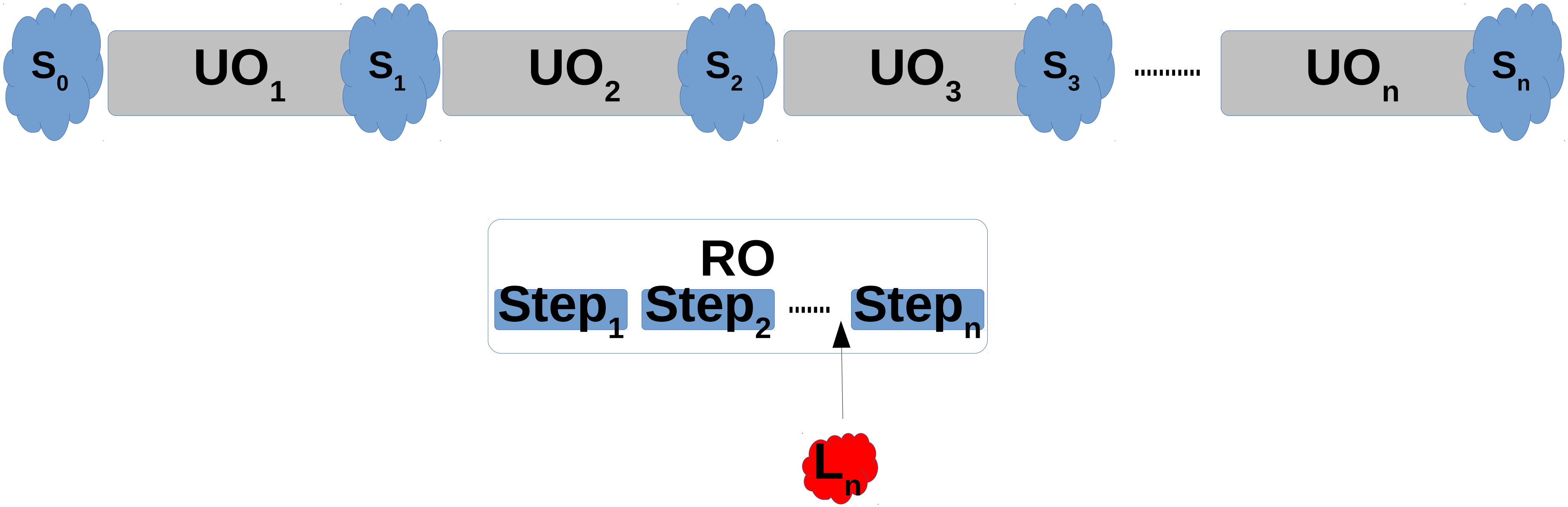
$S_0$ $UO_1$ $S_1$ $UO_2$ $S_2$ $UO_3$ $S_3$ .......... $UO_n$ $S_n$



RO

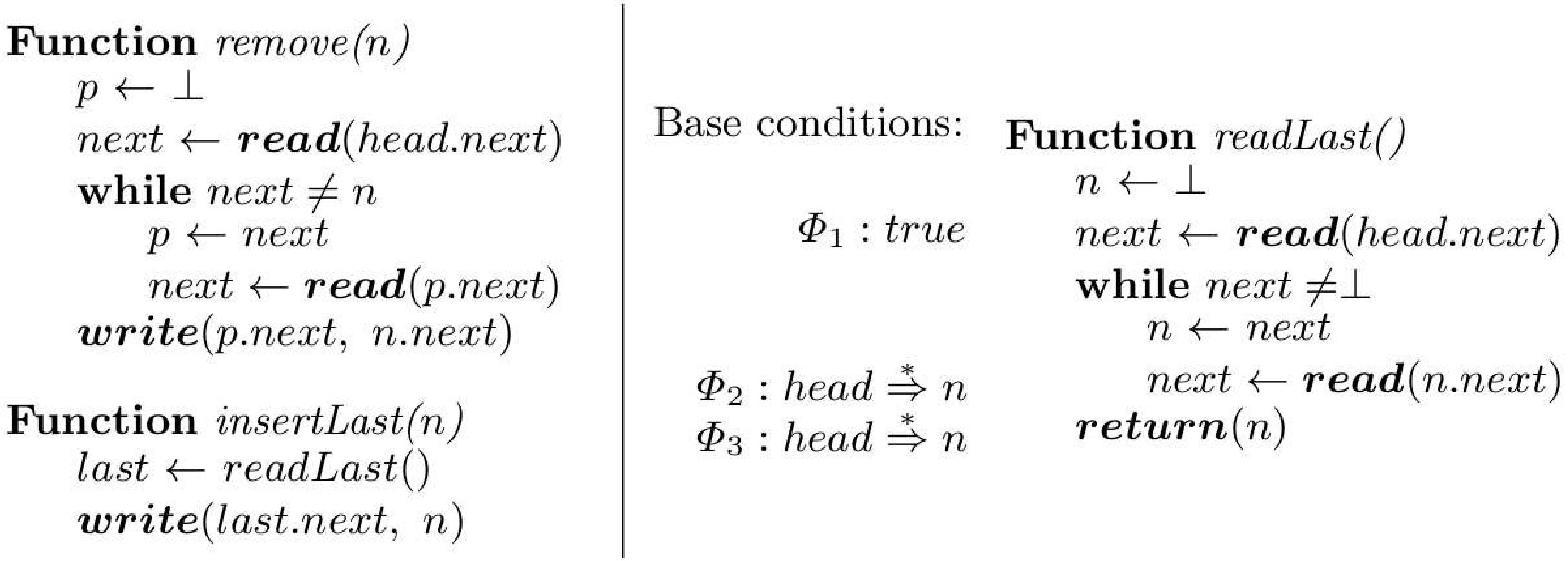
$Step_1$ $Step_2$ ....... $Step_n$

# Regularity



- Acceptable base points for RO's return step are only $S_1$, $S_2$, $S_3$.
  - Observes either the last update or a concurrent update.

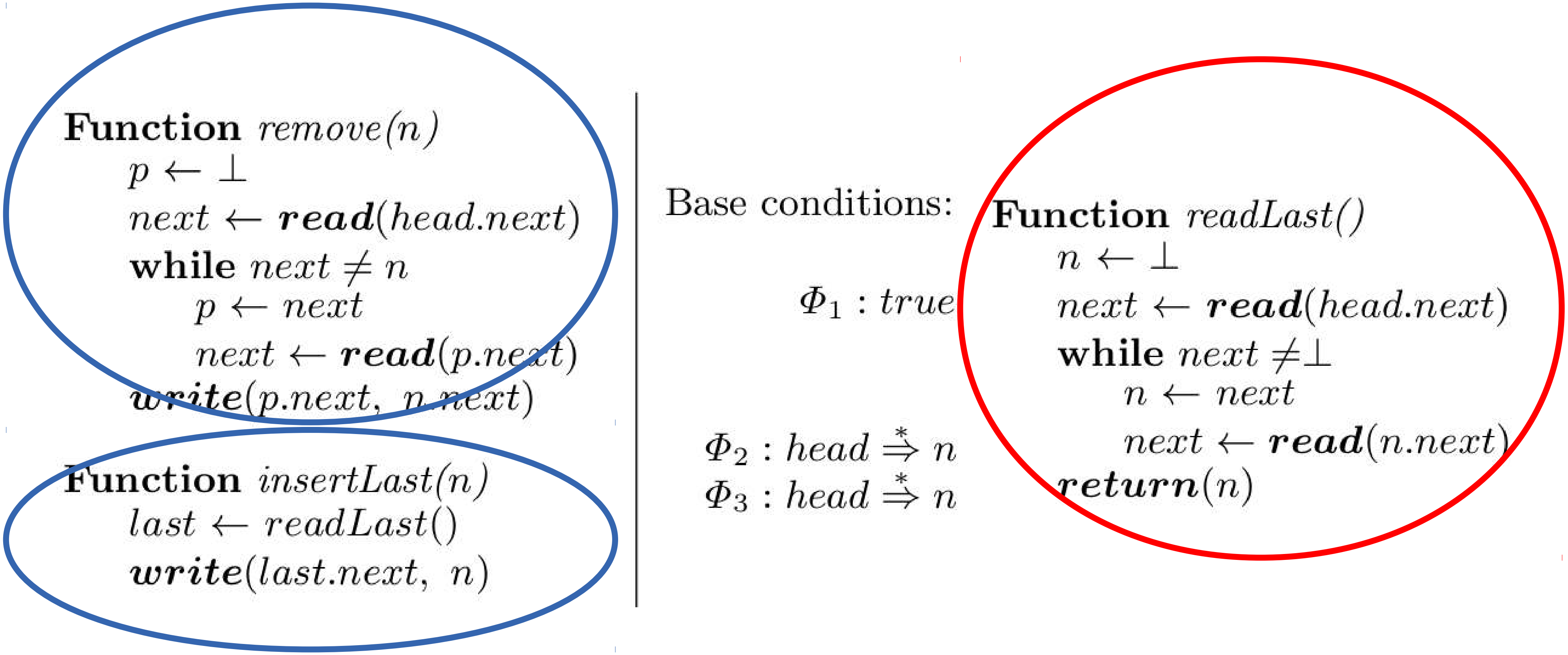
# Example



```
Function remove(n)
    $p \leftarrow \bot$
    $next \leftarrow \textbf{read}(head.next)$
    while $next \neq n$
        $p \leftarrow next$
        $next \leftarrow \textbf{read}(p.next)$
    $\textbf{write}(p.next,\ n.next)$

Function insertLast(n)
    $last \leftarrow readLast()$
    $\textbf{write}(last.next,\ n)$
```

```
Base conditions:                      Function readLast()
                                          $n \leftarrow \bot$
             $\Phi_1 : true$              $next \leftarrow \textbf{read}(head.next)$
                                          while $next \neq \bot$
                                              $n \leftarrow next$
  $\Phi_2 : head \overset{*}{\Rightarrow} n$          $next \leftarrow \textbf{read}(n.next)$
  $\Phi_3 : head \overset{*}{\Rightarrow} n$      $\textbf{return}(n)$
```

# Example



```
Function remove(n)
    $p \leftarrow \perp$
    $next \leftarrow$ read$(head.next)$
    while $next \neq n$
        $p \leftarrow next$
        $next \leftarrow$ read$(p.next)$
    write$(p.next,\ n.next)$

Function insertLast(n)
    $last \leftarrow readLast()$
    write$(last.next,\ n)$
```

Base conditions:

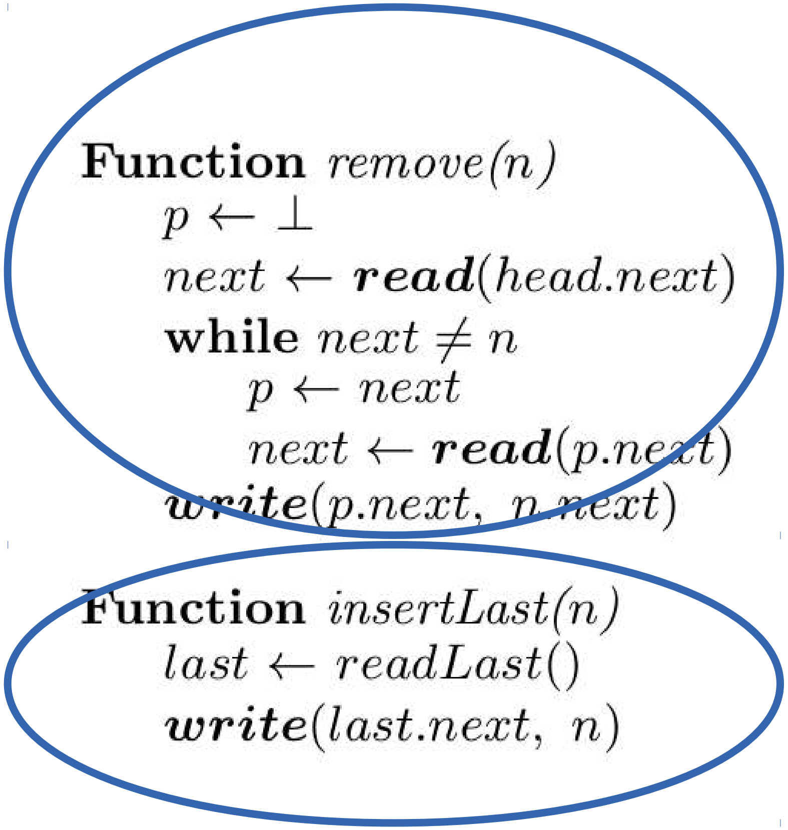
$\Phi_1 : true$

$\Phi_2 : head \overset{*}{\Rightarrow} n$

$\Phi_3 : head \overset{*}{\Rightarrow} n$

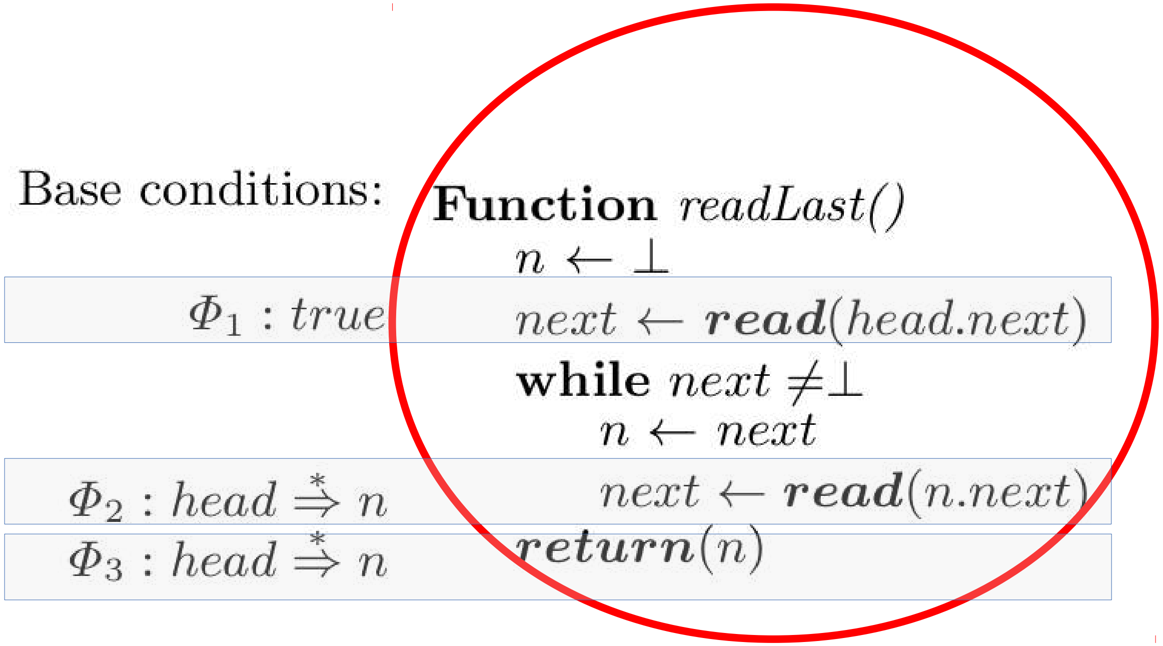
```
Function readLast()
    $n \leftarrow \perp$
    $next \leftarrow$ read$(head.next)$
    while $next \neq \perp$
        $n \leftarrow next$
        $next \leftarrow$ read$(n.next)$
    return$(n)$
```

# Example



```
Function remove(n)
    $p \leftarrow \perp$
    $next \leftarrow$ read($head.next$)
    while $next \neq n$
        $p \leftarrow next$
        $next \leftarrow$ read($p.next$)
    write($p.next$, $n.next$)
```

```
Function insertLast(n)
    $last \leftarrow readLast()$
    write($last.next$, $n$)
```



Base conditions:

- $\Phi_1 : true$
- $\Phi_2 : head \overset{*}{\Rightarrow} n$
- $\Phi_3 : head \overset{*}{\Rightarrow} n$

```
Function readLast()
    $n \leftarrow \perp$
    $next \leftarrow$ read($head.next$)
    while $next \neq \perp$
        $n \leftarrow next$
        $next \leftarrow$ read($n.next$)
    return($n$)
```

Where is the Problem?

It covers only single-writer designs

It does not cover composable designs

Can we cover a wider set?

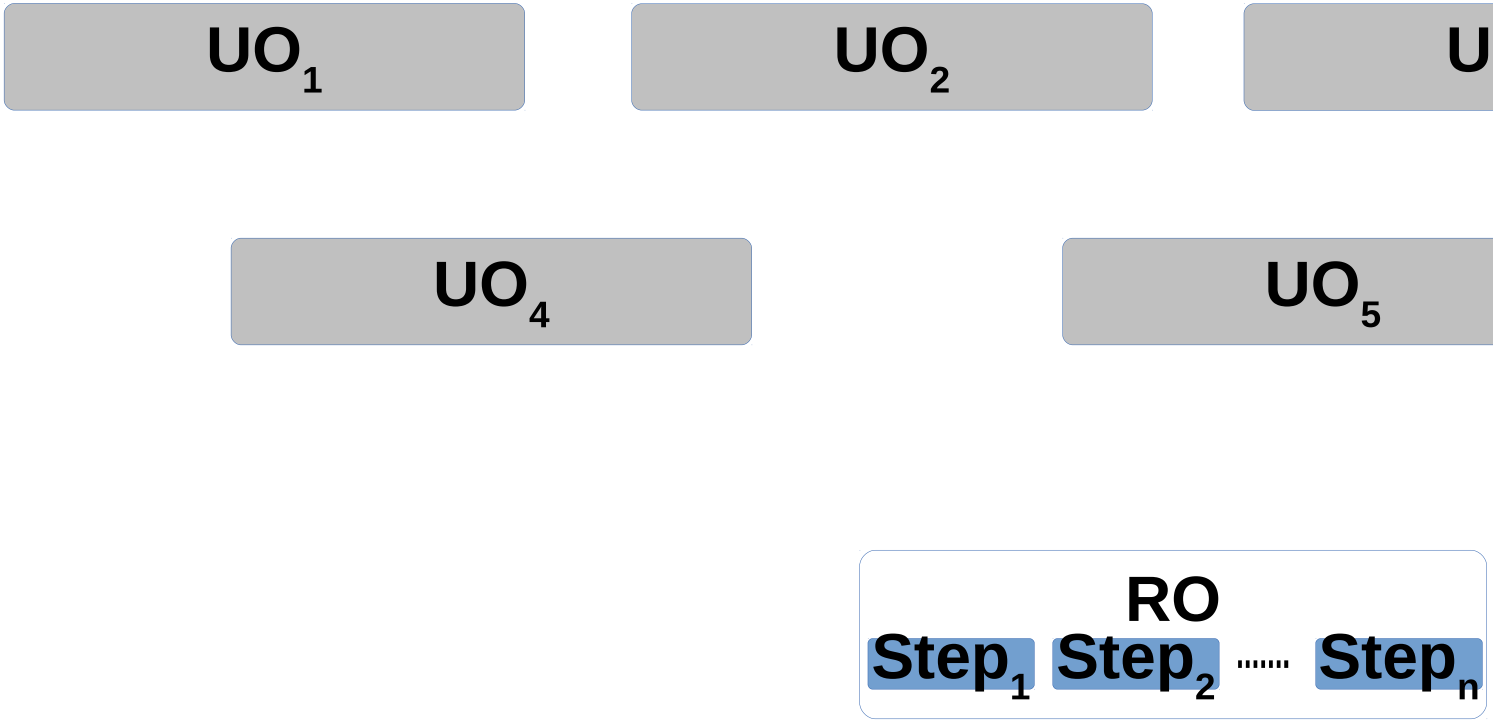
Optimistic Composable Data Structures

# Our Models

Single Writer Commit (SWC)

Composable SWC (C-SWC)

# SWC Model



$UO_1$

$UO_2$

$UO_3$

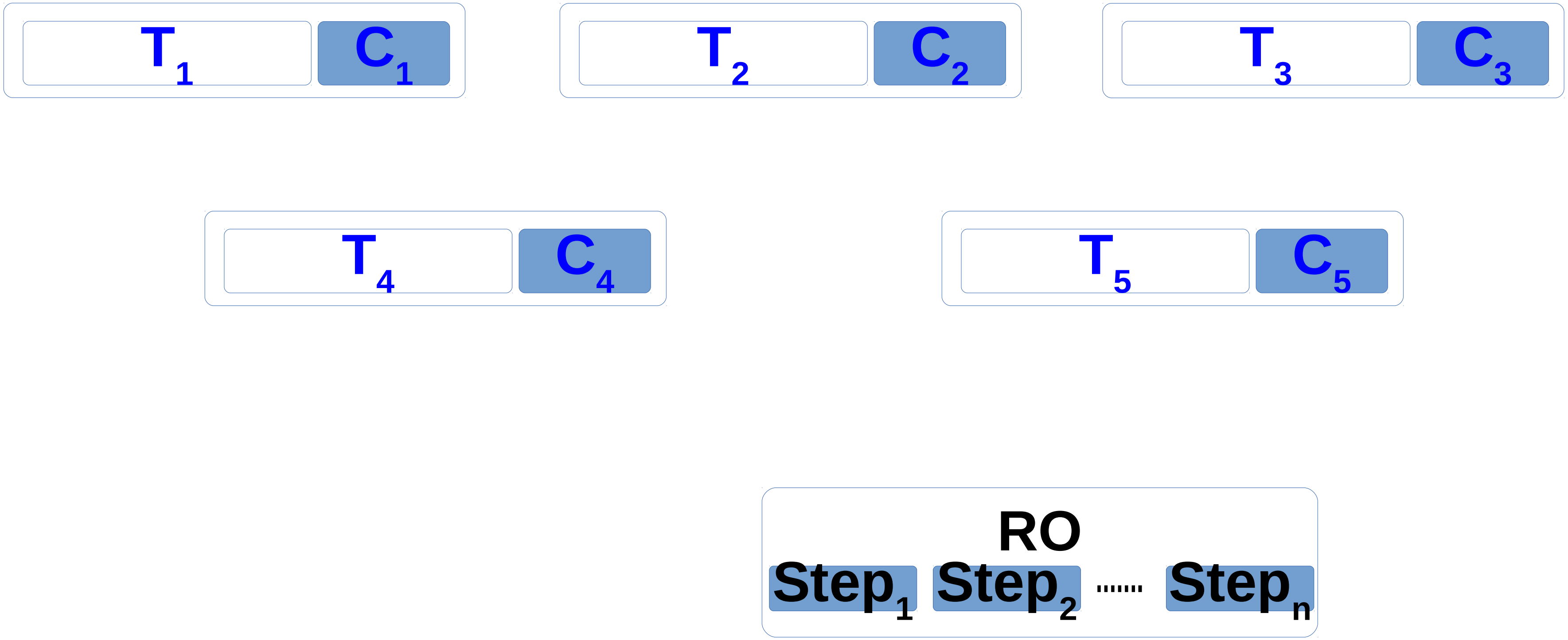
$UO_4$

$UO_5$

RO

$Step_1$ $Step_2$ ……. $Step_n$

# SWC Model



$T_1$ $C_1$

$T_2$ $C_2$

$T_3$ $C_3$

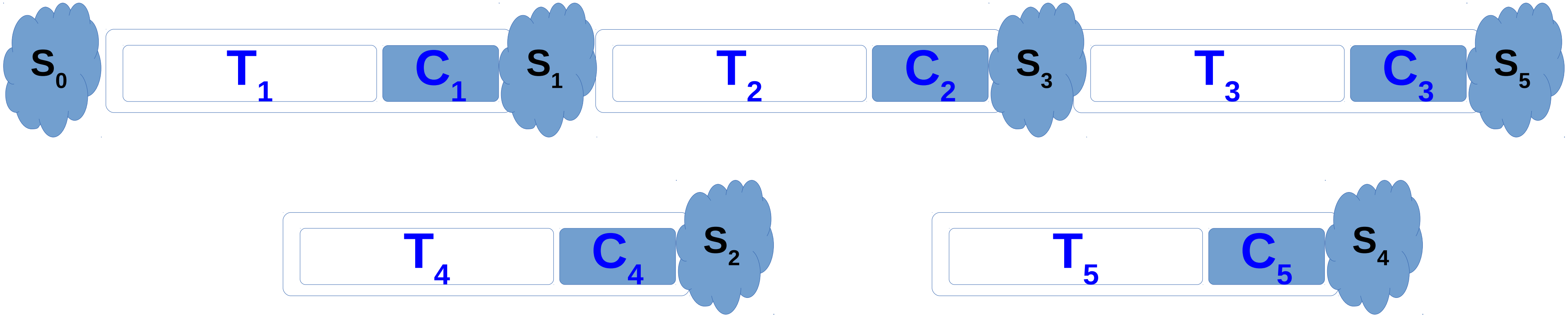
$T_4$ $C_4$

$T_5$ $C_5$

RO

$Step_1$ $Step_2$ ....... $Step_n$

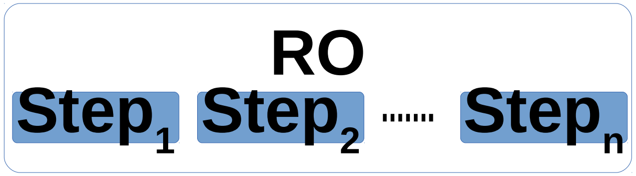
# SWC Model



$S_0$ $T_1$ $C_1$ $S_1$ $T_2$ $C_2$ $S_3$ $T_3$ $C_3$ $S_5$

$T_4$ $C_4$ $S_2$

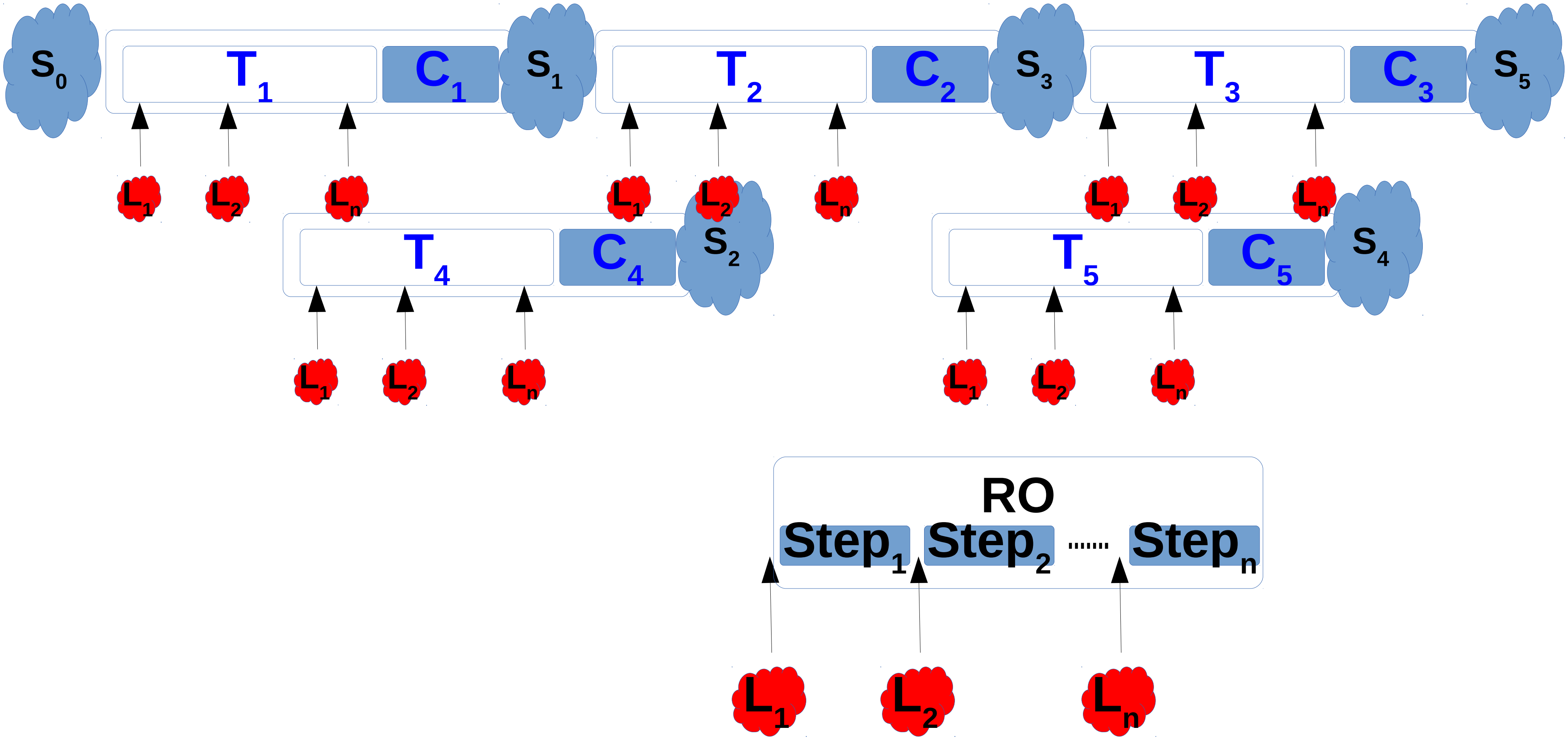
$T_5$ $C_5$ $S_4$



RO

$Step_1$ $Step_2$ ……. $Step_n$

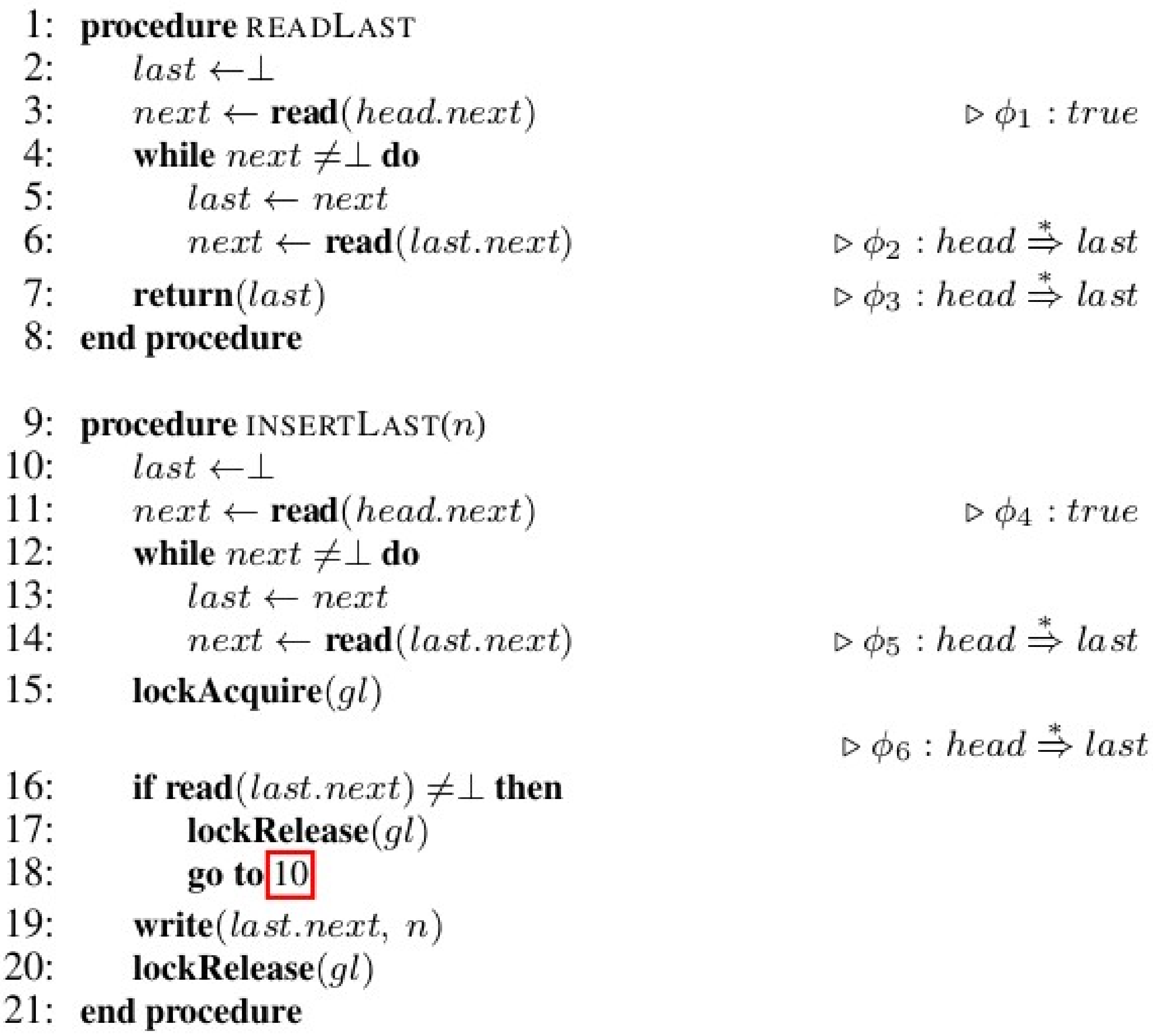
# SWC Model


$S_0$
$T_1$
$C_1$
$S_1$
$T_2$
$C_2$
$S_3$
$T_3$
$C_3$
$S_5$
$L_1$
$L_2$
$L_n$
$L_1$
$L_2$
$L_n$
$L_1$
$L_2$
$L_n$
$T_4$
$C_4$
$S_2$
$T_5$
$C_5$
$S_4$
$L_1$
$L_2$
$L_n$
$L_1$
$L_2$
$L_n$
RO
$Step_1$
$Step_2$
.......
$Step_n$
$L_1$
$L_2$
$L_n$

# Even More...

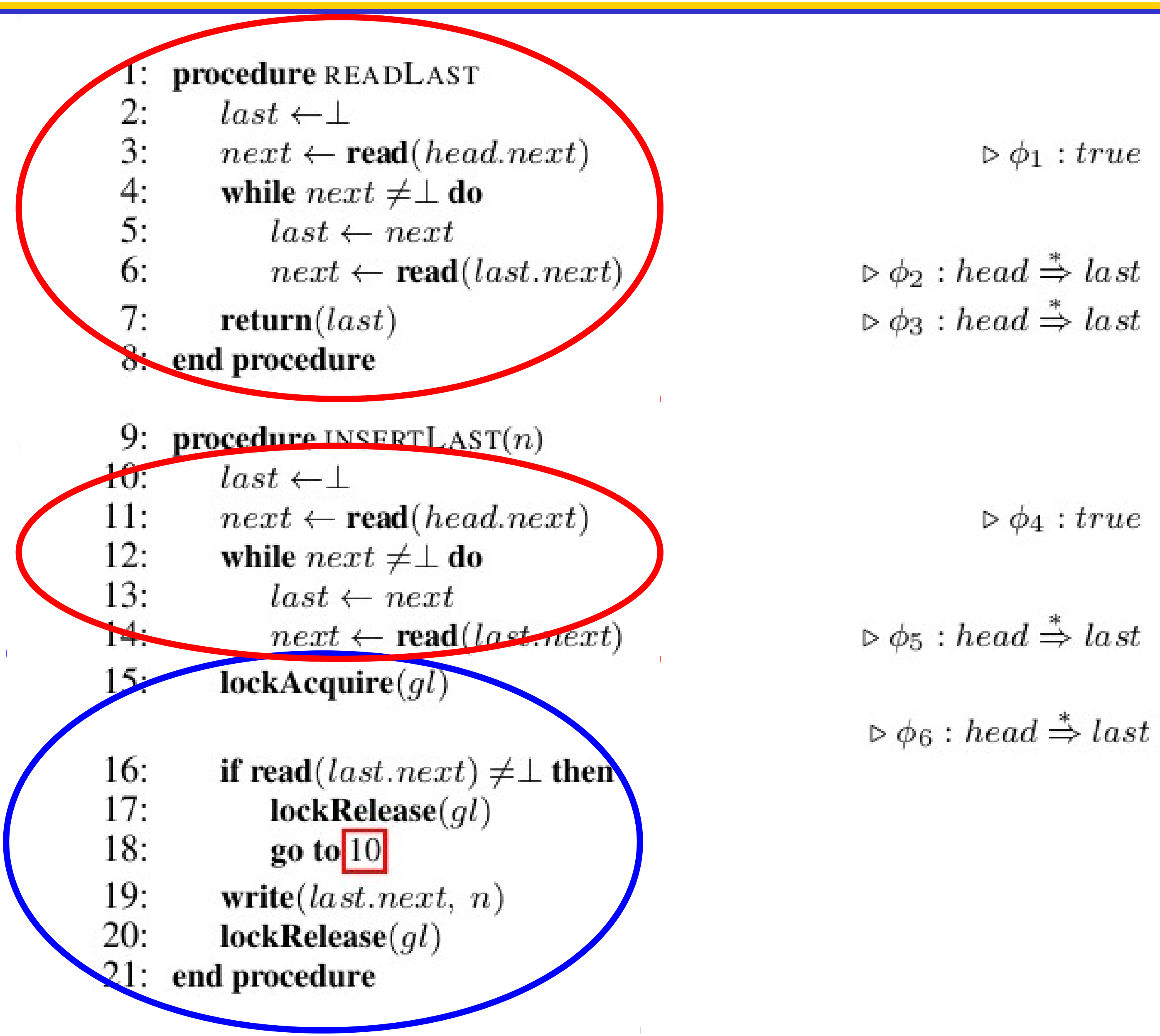
- Do we really need single commit at a time:
  - NO!!!
  - It is enough to execute commit phases atomically with *single lock atomicity (SLA)* guarantees.
  - More practical alternatives:
    - HTM (e.g. Intel TSX).
    - STM (e.g. NOrec “the SLA version”).

# Example



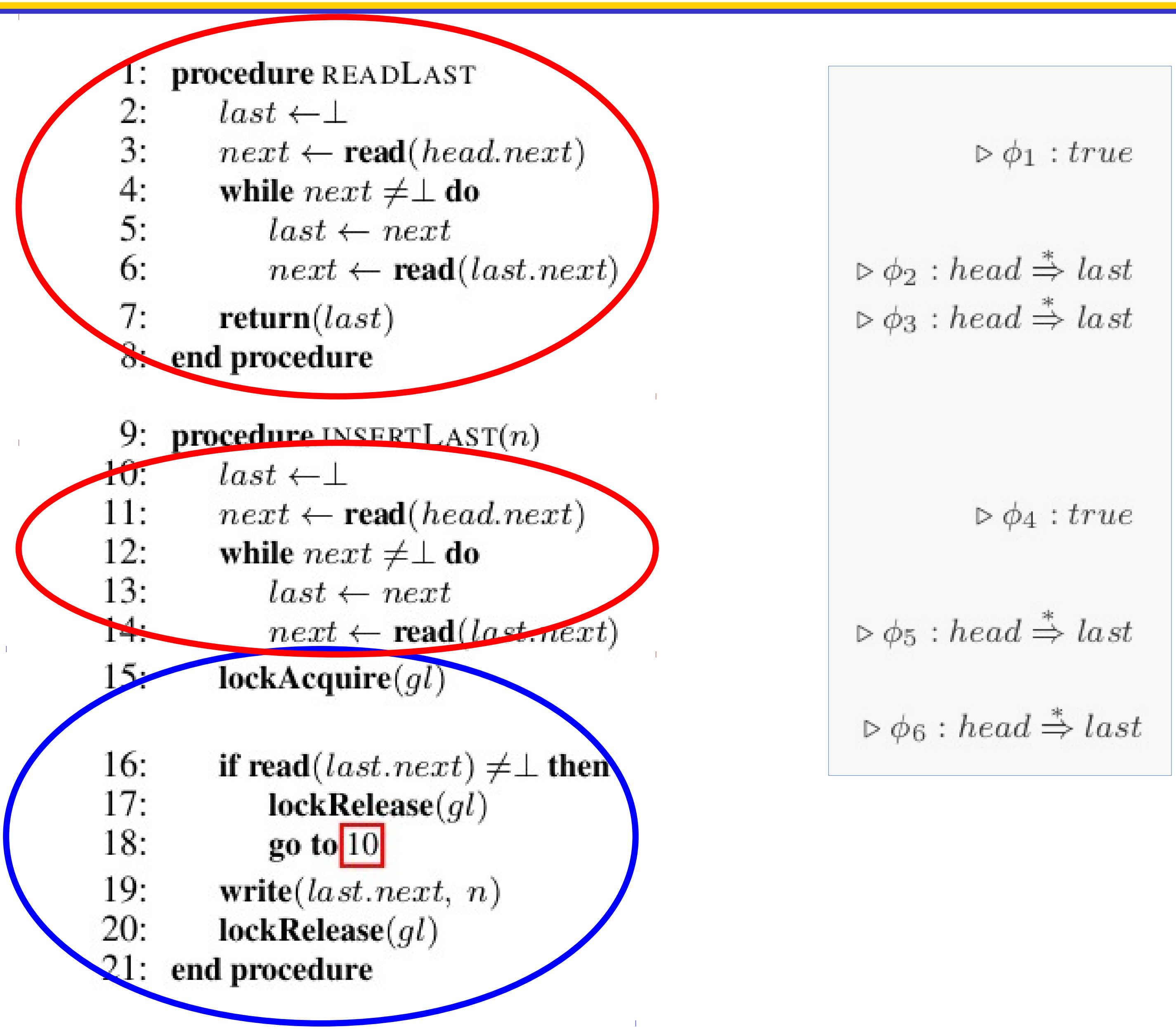
```
 1: procedure READLAST
 2:     $last \leftarrow \perp$
 3:     $next \leftarrow$ read$(head.next)$                       ▷ $\phi_1 : true$
 4:     while $next \neq \perp$ do
 5:         $last \leftarrow next$
 6:         $next \leftarrow$ read$(last.next)$           ▷ $\phi_2 : head \overset{*}{\Rightarrow} last$
 7:     return$(last)$                                ▷ $\phi_3 : head \overset{*}{\Rightarrow} last$
 8: end procedure

 9: procedure INSERTLAST$(n)$
10:     $last \leftarrow \perp$
11:     $next \leftarrow$ read$(head.next)$                       ▷ $\phi_4 : true$
12:     while $next \neq \perp$ do
13:         $last \leftarrow next$
14:         $next \leftarrow$ read$(last.next)$           ▷ $\phi_5 : head \overset{*}{\Rightarrow} last$
15:     lockAcquire$(gl)$
                                                      ▷ $\phi_6 : head \overset{*}{\Rightarrow} last$
16:     if read$(last.next) \neq \perp$ then
17:         lockRelease$(gl)$
18:         go to $\boxed{10}$
19:     write$(last.next,\ n)$
20:     lockRelease$(gl)$
21: end procedure
```

# Example



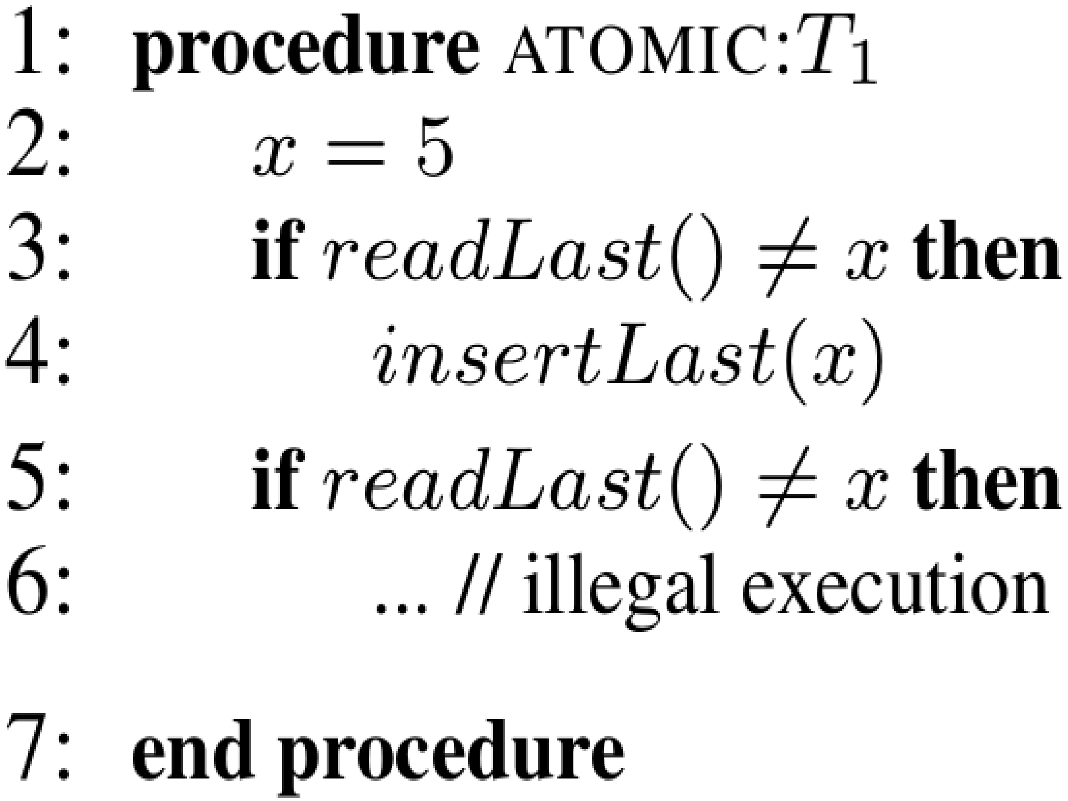
```
 1: procedure READLAST
 2:     $last \leftarrow \perp$
 3:     $next \leftarrow$ read$(head.next)$                ▷ $\phi_1 : true$
 4:     while $next \neq \perp$ do
 5:         $last \leftarrow next$
 6:         $next \leftarrow$ read$(last.next)$           ▷ $\phi_2 : head \overset{*}{\Rightarrow} last$
 7:     return$(last)$                               ▷ $\phi_3 : head \overset{*}{\Rightarrow} last$
 8: end procedure

 9: procedure INSERTLAST$(n)$
10:     $last \leftarrow \perp$
11:     $next \leftarrow$ read$(head.next)$                ▷ $\phi_4 : true$
12:     while $next \neq \perp$ do
13:         $last \leftarrow next$
14:         $next \leftarrow$ read$(last.next)$           ▷ $\phi_5 : head \overset{*}{\Rightarrow} last$
15:     lockAcquire$(gl)$
                                                     ▷ $\phi_6 : head \overset{*}{\Rightarrow} last$
16:     if read$(last.next) \neq \perp$ then
17:         lockRelease$(gl)$
18:         go to $\boxed{10}$
19:     write$(last.next,\ n)$
20:     lockRelease$(gl)$
21: end procedure
```

# Example



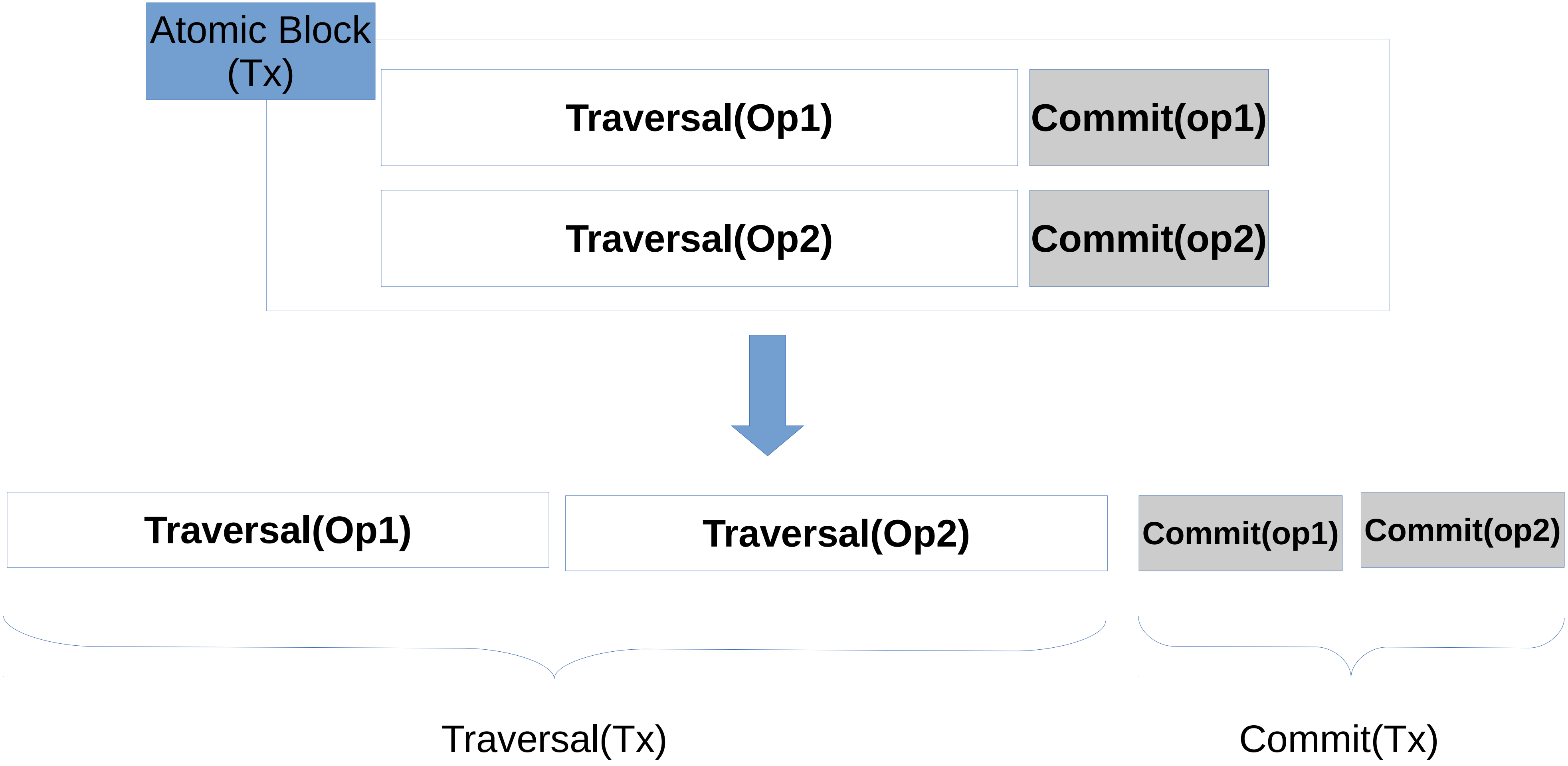
```
1:  procedure ReadLast
2:      $last \leftarrow \perp$
3:      $next \leftarrow$ read$(head.next)$                 $\triangleright\ \phi_1 : true$
4:      while $next \neq \perp$ do
5:          $last \leftarrow next$
6:          $next \leftarrow$ read$(last.next)$         $\triangleright\ \phi_2 : head \overset{*}{\Rightarrow} last$
7:      return$(last)$                                 $\triangleright\ \phi_3 : head \overset{*}{\Rightarrow} last$
8:  end procedure

9:  procedure InsertLast$(n)$
10:     $last \leftarrow \perp$
11:     $next \leftarrow$ read$(head.next)$                 $\triangleright\ \phi_4 : true$
12:     while $next \neq \perp$ do
13:         $last \leftarrow next$
14:         $next \leftarrow$ read$(last.next)$         $\triangleright\ \phi_5 : head \overset{*}{\Rightarrow} last$
15:     lockAcquire$(gl)$
                                                       $\triangleright\ \phi_6 : head \overset{*}{\Rightarrow} last$
16:     if read$(last.next) \neq \perp$ then
17:         lockRelease$(gl)$
18:         go to $\boxed{10}$
19:     write$(last.next,\ n)$
20:     lockRelease$(gl)$
21: end procedure
```

# Composable SWC Model (C-SWC)



```
procedure ATOMIC:$T_1$
    $x = 5$
    if $readLast() \neq x$ then
        $insertLast(x)$
    if $readLast() \neq x$ then
        ... // illegal execution
end procedure
```

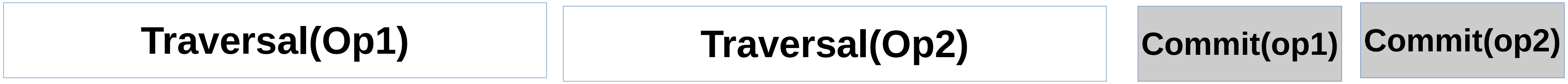
# Composable SWC Model (C-SWC)



# What is remaining?

- Internal Consistency.
  - The commit phase of each operation reflects what the operation observed in its traversal.
  - The shared state of an operation is visible to subsequent operations in the same transaction.

# How to prove internal consistency?



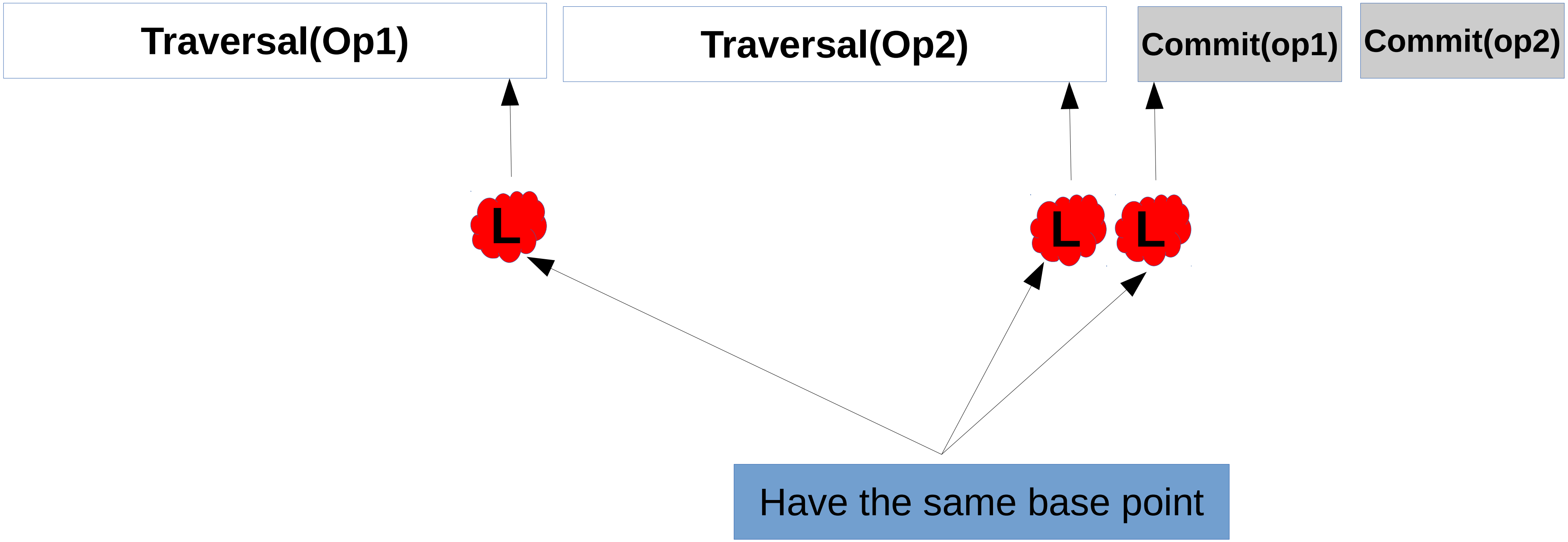
Traversal(Op1)

Traversal(Op2)

Commit(op1)

Commit(op2)

# How to prove internal consistency?



Traversal(Op1)

Traversal(Op2)

Commit(op1)

Commit(op2)

L

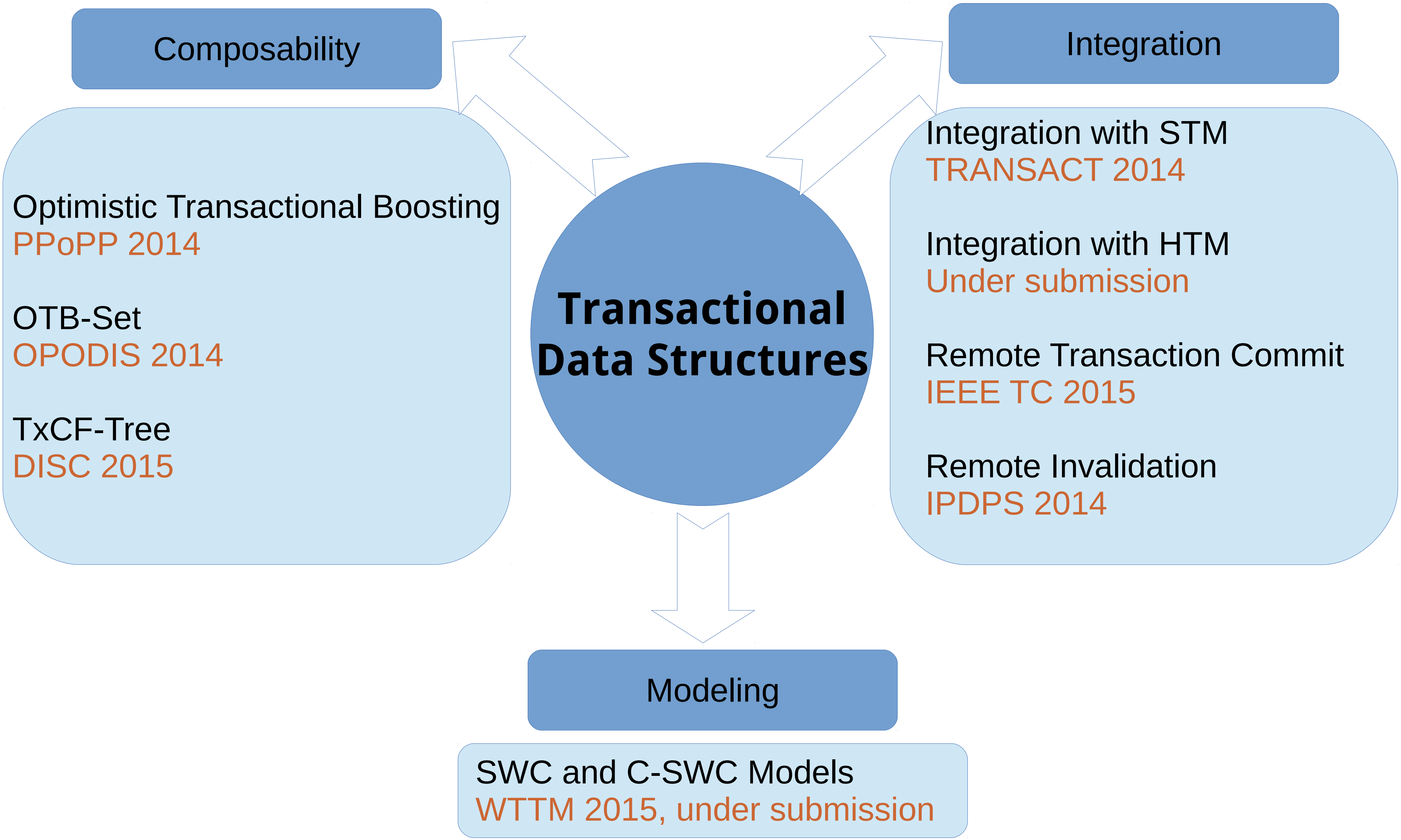
L

L

Have the same base point

# Conclusions

# Our Contributions



Transactional Data Structures

Composability

Optimistic Transactional Boosting
PPoPP 2014

OTB-Set
OPODIS 2014

TxCF-Tree
DISC 2015

Integration

Integration with STM
TRANSACT 2014

Integration with HTM
Under submission

Remote Transaction Commit
IEEE TC 2015

Remote Invalidation
IPDPS 2014

Modeling

SWC and C-SWC Models
WTTM 2015, under submission

# Thanks!

Questions?